NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

B80448

ADA® AS A PAEDEUTIC TOOL FOR ABSTRACT DATA TYPES

by

Richard Neely Britnell

December 1988

Thesis Advisor: C. Thomas Wu

Approved for public release; distribution is unlimited

UNCLASSIFIED

RITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

EPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b RESTRICTIVE MARKINGS
ECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
DECLASSIFICATION/DOWNGRADING SCHEDULE		
ERFORMING ORGANIZATION REPORT NUMBER(S)		5 MONITORING ORGANIZATION REPORT NUMBER(S)
NAME OF PERFORMING ORGANIZATION val Postgraduate School	6b. OFFICE SYMBOL *(If applicable)* Code 52	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
ADDRESS *(City, State, and ZIP Code)* nterey, California 93943-5000		7b. ADDRESS *(City, State, and ZIP Code)* Monterey, California 93943-5000
NAME OF FUNDING/SPONSORING ORGANIZATION	8b OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
ADDRESS *(City, State, and ZIP Code)*		10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.

TITLE *(Include Security Classification)*

ADA AS A PAEDEUTIC TOOL FOR ABSTRACT DATA TYPES

PERSONAL AUTHOR(S)

Britnell, Richard N.

TYPE OF REPORT	13b TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15 PAGE COUNT
ster's Thesis	FROM ______ TO ______	1988, December	152

SUPPLEMENTARY NOTATION

he views expressed in this thesis are those of the author and do not reflect the official olicy or position of the Department of Defense or the U.S. Government.

COSATI CODES			18 SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	Abstract Data Type; Ada; Strong Typing; Generic; Inheritance; Information Hiding; Exception; Separate Compilation

ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This thesis discusses the pedagogy for abstract data types (ADTs). nguage features needed for teaching ADTs are listed and arguments for need- g them are provided. ADTs are implemented in Ada to show the benefit of ese features. Ada possesses the desired language features but the inherit- ce provided in Ada is limited. ADT interface considerations and ADT imple- ntation design strategies are critical to the pedagogy for ADTs and are also scussed. Although Ada is complex and difficult to learn and it only provides mited inheritance, it is an excellent language for teaching ADTs.

DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
a. NAME OF RESPONSIBLE INDIVIDUAL Prof. C. Thomas Wu	22b TELEPHONE *(Include Area Code)* (408) 646-3391 — 22c OFFICE SYMBOL Code 52Hq

D FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete — SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

Ada as a Paedeutic Tool for Abstract Data Types

by

Richard Neely Britnell
Lieutenant Commander, United States Navy
B.S., Auburn University, 1976

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1988

ABSTRACT

This thesis discusses the pedagogy for abstract data types (ADTs). Language features needed for teaching ADTs are listed and arguments for needing them are provided. ADTs are implemented in Ada to show the benefit of these features. Ada possesses the desired language features but the inheritance provided in Ada is limited. ADT interface considerations and ADT implementation design strategies are critical to the pedagogy for ADTs and are also discussed. Although Ada is complex and difficult to learn and it only provides limited inheritance, it is an excellent language for teaching ADTs.

DISCLAIMER

Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

TABLE OF CONTENTS

I. BACKGROUND

The primary objective of this thesis is to determine the language features needed in a language for teaching abstract data types (ADTs). Additionally the thesis will show the benefits of choosing a language with these features for teaching ADTs. The major emphasis of ADT instruction should be the concept of abstraction. A language for teaching ADTs should therefore support abstraction well. Ada is such a language. The ADT increases the level of abstraction for a programmer and reduces the complexity of a problem. This is achieved by defining certain operations on a data type which is also defined by the programmer. These operations can then be used by the programmer without concerning himself with the details of how they are performed. Programmers should be concerned only with these operations and how to use them. They should not be concerned with a list of DON'Ts designed to prevent the programmer from violating the ADT.

An ADT is composed of two parts, the set of data values and the primitive operations on those data values. [Ref. 1:p. 184] A stack is a classic example of an ADT. The stack could hold any data type, such as an integer or record. The operations on the stack might contain Pop, Push, Top, Full, Empty, Create and Destroy. ADTs are useful when programming because they provide a higher level of abstraction that is more easily understood and used by the programmer. When programmers build their own abstract data types, they have detailed knowledge of how the set of operations work and detailed knowledge of the underlying data structure. Additionally the level of abstraction is raised above that provided by the programming language that the programmer is using.

Computers of the 1940's and 50's required that programs be hard wired for the particular computer. With the advent of high-level languages the level of abstraction was raised significantly. The data and operations of a problem could now be stated using the data and control structures of a language. A greater variety of control and data structures became available in the 1970s in languages such as Pascal. Programmers still relied mainly on translating the problem directly into the structures provided by the language. "Modular software construction and object-oriented design supports the second major jump in abstraction that is possible in the software development process." [Ref. 2:p. 30] The programmer can create his own abstractions with abstract data types and escape the relatively limited level of abstraction provided by the data and control structures of a language. [Ref. 2:p. 30]

The major advantage of ADTs is the higher level of abstraction that they provide to the programmer. ADTs achieve this higher level of abstraction by separating the essential qualities of data, their structure and operations, from the inessential details of their representation and implementation. [Ref. 3:pp. 26-27] The programmer's use of the ADT should be restricted to the well defined set of operations. Adhering to this restriction will preserve the integrity of the data structure. Violating this restriction destroys the data structure's integrity and circumvents the advantage of the higher level of abstraction.

Undisciplined programmers by design and by chance violate ADTs. When learning about ADTs, programmers must understand the concept of an ADT first and foremost. Once this understanding is achieved and accepted, the programmer must then use his own self-discipline to prevent violations by design when the programming language in use allows the ADT to be violated. But violations by chance may still occur if the language in use fails to have sufficient safety features built into it to prevent these violations.

The language a programmer is using provides some safety features that prevent violating the ADTs built into the language. These safety features include typing, coercions, and information hiding. ADTs built by the programmer should have similar features to prevent the corruption of the ADT. Often times this is not possible as the language being used may not provide or allow the programmer to build the necessary safety features to prevent corruption.

The implementation of built-in ADTs in Pascal is hidden from the programmer. For example, the underlying machine representation of an integer is unknown to the programmer. Having this information hidden prevents the programmer from manipulating it. ADTs written in Pascal by a programmer however are inside a glass house that the programmer has built. The programmer knows how the data is stored and manipulated. Global data structures that should be manipulated only by procedure and function calls, that comprise the well defined set of operations, can be accessed directly from any portion of the program. The programmer must be cautious not to do this, as this is a major pitfall.

Another potential pitfall involves interface design and specification. Programmers have difficulty with designing the interface for an ADT. Often times the interface's design is affected by the underlying data structure that the programmer intends to use. This can cause major problems when the underlying data structure is changed. The interface should be designed so that underlying data structure changes do not require interface changes. The abstraction provided by the operations of the ADT should be preserved even if the underlying implementation changes.

The operations allowed on the ADT are placed in the interface. Pre- and post-conditions for each operation should be specified. The operations that are allowed should be carefully considered. "If the operation set is not expressive enough, it might be impossible or inconvenient to implement certain useful functions...." [Ref. 4:p. 33]

ADTs when designed well and used properly can be of great benefit to programmers. Conversely, when they are not designed well or when used improperly, they can create major problems for the programmer. The language for implementing ADTs should support abstraction well.

ADTs have been implemented in Ada and are in the appendices. Ada was chosen because it supports abstraction well through the use of packages. The language contains features that prevent some of the common actions that corrupt ADTs. Ada also provides a capability for writing generic ADTs. Generic capability extends the level of abstraction that can be achieved. The ADT implementations found in the appendices will be referred to in Chapters II, III and IV.

Chapter II will discuss the built in language features required of an implementation language for constructing robust ADTs. These features will include those that support the concept of an ADT.

Chapter III will explain what should be provided in the interface between the ADT and the user and how the interface should be built. This chapter will also discuss the user interface changes caused by different implementations of an ADT.

Chapter IV will discuss single ADT implementation design strategies as well as implementation design strategies for multiple ADTs that are related or dependent on one another. The importance of the material in the previous two chapters will be amplified by the discussion of these strategies.

Chapter V will summarize the most important points developed. Recommendations will be made that fall into two categories, those concerning teaching ADTs and those concerning Ada.

II. LANGUAGE FEATURES NEEDED FOR TEACHING ABSTRACT DATA TYPES

The implementation language for an ADT should provide as many features as possible that enhance the ADT concept and also provides safety mechanisms that prevent violations of the ADT. The implementation language should provide the programmer the following features:

1. Mechanisms for constructing generic programs.
2. Separate compilation.
3. Constructs that allow and enforce information hiding.
4. Strong typing.
5. Inheritance.
6. Exception handling.

Ada provides all of these features but the inheritance is somewhat limited.

A. GENERICS

Generics embody the pure abstraction desired of an ADT. A stack that contains integers may have the two operations of Pop and Push. These operations affect two data types namely the integer and the stack of integers. A stack of records is identical in concept to the stack of integers with the exception of the data type being stacked, records in place of integers. The operations on a stack are related to the concept of a stack not to the data type being stored in the stack. A generic stack capable of storing any data type specified by the programmer allows the operations of the stack to be fully abstracted and provides a stack independent of the data to be stored.

A language should support generic ADTs. As explained above, an ADT should be implementable independent of the data type manipulated by the ADT. The operation set of the ADT should be applicable to any data type. Generic ADTs allow the operations to be abstracted regardless of the data type to be manipulated.

If logical parallels between real world objects and ADTs can be shown when teaching ADTs then the student is more likely to understand the concept of an ADT because he already understands a real world object. Examples of lists in the real world are grocery lists, mailing lists, and "things to do" lists. Each of these lists contain different objects; groceries, addresses, and jobs. But all of the lists share the concept of items enumerated for a specific reason and also the things (operations) that can be done with a list. The list concept is separate from the items in the list. Each of the examples above are specific instances of a list. A list ADT should only embody the list concept, which is a concept that the student is already familiar with. A language that allows generic ADTs will support this logical comparison between the ADT and its real world counterpart.

Ada provides for generic programs with the generic package or subprogram (i.e., procedures and functions). The generic program unit serves as the template from which actual packages or subprograms can be instantiated. When instantiating a specific instance of the generic template the programmer must provide the actual parameters that will take the place of the generic parameters in the template. [Ref. 5: p. 14] The generic linked list found in Appendix A must have the generic parameter Item specified by the programmer when instantiating the package.

The parameter Item can be any language or user defined type. This gives the user of the generic ADT many different specific ADTs. He can have a linked list of integers, characters, records, or linked lists. The added benefit is that the proof of correctness of the specific ADTs can be implied from the proof of correctness of the generic ADT.

B. SEPARATE COMPILATION

ADTs are often used in large software projects because they allow the breakdown of the overall project into smaller modules which can then be divided among various programmers. Programmers working on different modules may have the need to use some of the operations provided by an ADT written by a different programmer or team. If modules can be compiled separately then the programmers can make use of each others work prior to project completion. Separate compilation helps realize the full benefits of this break down and encapsulation.

Ideally a language should provide for separate compilation of the implementation and the specification of an ADT. This feature allows the operations of ADTs to be fully specified and the ADT made "available" for use prior to its implementation. Changes to an ADT's operation set can be made easily when only the specification exists, thereby providing greater flexibility to the design of a large project consisting of several ADTs. The implementation of a project can be simplified because the complete specification of the ADTs can be completed before the ADTs are implemented.

Students can focus their attention solely on implementing an ADT when separate compilation is available. Concern with the interactions between the ADT and the program that uses it can be put aside while the ADT is built. Separate compilation also supports the programming team methodology. An instructor can specify the interface of the ADT and once the student has implemented the ADT, the instructor can then provide a driver test program to test the implementation built by the student.

Some objects in the real world that are made into ADTs are viewed by students as independent stand alone objects. A list does not require anything external to itself to be a list. Therefore a list ADT should not require anything external either. An encoded list ADT should be compilable separate from the program(s) that will use it. This provides the student an abstract concept in code that can be used at any time, just as he could use the object in the real world. Students can more clearly see the elegance an ADT provides if the ADT can be separately compiled. Additionally, the specification and implementation of an ADT should be independent as long as the implementation fulfills the requirements of the specification. If the specification and implementation can also be compiled separately, the student can see that the operations of an ADT are independent of the implementation. A written list and a mental list are two different implementations of a list. The operations on a list though remain the same regardless of the implementation.

Ada provides for separate compilation of program units. More importantly Ada allows a program unit specification to be compiled into a library unit. This defines the interface between the program unit and the rest of the program. An example of a specification for a Linked List ADT can be found in Figure 1 with further details of the specification found in Appendix A. The program unit body can then be compiled into a secondary unit. The body contains the executable code of the corresponding library unit. [Ref. 6:p. 222]

```
Generic
   Type Item is Private;

Package Generic_List is
   Type List is Limited Private;

BEYOND_END : Exception;
NOT_FOUND : Exception;
INSERT_BEYOND : Exception;
DELETE_OUT_OF_RANGE : Exception;

Procedure Clear (L : in out List);

Function Full (L : in List) Return Boolean;

Function Empty (L : in List) Return Boolean;

Procedure Insert (L : in out List; P : in Integer;
                                   I : in Item);

Procedure Delete (L : in out List; P : in Integer;
                                   I : out Item);

Procedure Length (L : in List; Long : out Integer);

Procedure Find_Item (L : in List; P : in Integer;
                                   I : out Item);

Procedure Find_Pos (L : in List; P : out Integer;
                                   I : in Item);
Private
   Type Node;
   Type List is Access Node;
End Generic_List;
```

Figure 1. Specification of Linked List

The distinction between the specification and body of a program unit is significant. By defining the operations that an ADT will have, the programmer can create a library unit that is then visible to another compilation unit by means of a with clause. [Ref. 6:p. 222] This feature of Ada allows for a very modular design that can be defined and compiled into <u>library units</u> before the first line of executable code is written. [Ref. 2:p. 30] The design of the ADT can be completely abstracted from the implementation.

C. INFORMATION HIDING

The implementation details of an ADT should be known only to the author of the ADT. Programmers that use the ADT should be restricted to the predefined set of operations allowed by the ADT. An ADT's integrity will remain sound if the only operations performed are those specified by the author. The essence of information hiding is best captured in Parnas's Principles:

1. One must provide the intended user with all the information needed to use the module correctly and nothing more.

2. One must provide the implementor with all the information needed to complete the module and nothing more. [Ref. 1:p. 285]

Implementation details hidden from the user inhibit the user's ability to corrupt the ADT. Additionally the user's focus is restricted to what the ADT provides and not how it is provided. Therefore, his thought process for solving a problem is simplified by the higher level of abstraction that the ADT provides.

Students learning about ADTs are primarily novice programmers. They are still learning the tenets of good programming. When an error in a program is discovered the quickest fix is often the one chosen. When the program is retested and the error fails to reappear the student concludes that he has fixed the problem. The student's correction; however, may be violating the principle of information hiding. He may have solved the problem by directly manipulating the underlying data structure of the ADT that he is using, if the language fails to enforce information hiding. The student has now violated the ADT. The benefits that the ADT provide have been muddied by the student's actions. Most importantly, the student has failed to use the ADT properly and he may not even realize this fact. With information hiding enforcement provided by the language the student is prevented from making this kind of error.

Ada provides a very good mechanism for information hiding through packages and private types. The package is composed of two parts, the specification and the body. The specification formally defines the ADT and provides the interface to the outside world. The body contains the hidden details of the implementation. [Ref. 5:p. 13] A complete specification of a Stack ADT is provided in Figure 2. The user and author of the package should both be able to perform their jobs correctly by using only the information provided in a well documented specification.

Ada also enforces information hiding through <u>private</u> and <u>limited private</u> types. Both of these hide the implementation details of the user defined type. The former

```
Generic
   Type StackItem is Private;

Package Gen_Stack is
   Type Stack is Limited Private;


Procedure Clear (S : in out Stack);
-- pre - None.
-- post - S is an empty stack.

Function Full(S : in Stack) Return Boolean;
-- pre - None.
-- post - True if stack S can not have more items added,
--         otherwise False.

Function Empty(S : in Stack) Return Boolean;
-- pre - None.
-- post -True if stack S has no items in it, otherwise
   False.

Procedure Push (S : in out Stack; I : in StackItem);
-- pre - The size of S has not reached its maximum.
-- post - S has item I on top of it.

Procedure Pop (S : in out Stack; I : out StackItem);
-- pre - S is not empty.
-- post - Top of S-pre is assigned to I and S no longer
--         contains I.

Procedure Size (S : in Stack; Depth : out Integer);
-- pre - S exists.
-- post - Depth is equal to the number of items in S.

Procedure Top (S : in Stack; I : out StackItem);
-- pre - S is not empty.
-- post - I is the top item of S.  S is unchanged.


Private
   Type Plate;
   Type Stack is Access Plate;

end Gen_Stack;
```

Figure 2. Full Specification of Stack ADT

allows the assignment, equality and inequality operations to be performed between objects of the same type outside the package. The latter prohibits these three operations between objects of the same type. The operations available on a limited private type are restricted then to only those operations given in the specification part of the package. [Ref. 6:p. 158]

Ada also prohibits function side effects. Namely, a function can only return a value. Pascal fails to provide this capability and functions such as that shown in Figure 3 can occur as a result.

The problem with the Length function is that the current pointer's position in the list has been altered. The position of the current pointer prior to the function call has been lost. The current pointer is now pointing to nil at the very end of the list. The length has been provided but the underlying data structure, List record, has been altered. This alteration is a side effect of calling the Length function. Parameters passed to a function should not be altered and returned in the altered state. This feature also helps to enforce the security of the underlying data structure which is one of the goals of information hiding.

D. STRONG TYPING

Strong typing requires that whenever an object is used its type must match the type necessitated by the context. [Ref. 1:p. 192] Failure to provide strong typing can allow

```
Type
   NodePointer = ^ Node;
   Node = Record
             Content : Char;
             Next : NodePointer;
          end;

   List = Record
             Current : NodePointer;
             Head : NodePointer;
          end;
Var
   L : List;

Function Length (Var L : List) : Integer;
(* Returns the length of list L.*)
Var
  Count : Integer;

Begin
   Count := 0;

   L.Current := L.Head; (*Puts Current at front of
list*)
   While L.Current <> nil Do
      begin                                 (*Moves down the *)
         L.Current := L.Current^.Next;(*list one node *)

         Count := Count + 1;        (*at a time counting*)
      end;                                  (*each node.*)
   Length := Count;

End;
```

Figure 3. Example of Function Side-effect

operations to be performed between different data types that are illogical. By allowing illogical operations the abstraction achieved with an ADT becomes clouded and the ADT's underlying data structures become suspect.

Type checking can be completed statically (compile-time) or dynamically (run-time). The advantage of static checking is that type errors are found earlier than dynamic checking.

Strong typing can be achieved either statically or dynamically. [Ref. 1:p. 464] BUT if dynamic checking is done, then some type of protection should be provided in the language to control the action of the program if a type violation occurs.

Novice programmers often make mistakes related to type incompatibilities. Assignments, relational operations and parameter passing are prime opportunities for a type incompatibility error. Students need strong typing to catch these errors. The student could also become confused by inconsistent actions provided by ADT operations if strong typing were not enforced.

Ada performs type checking statically and uses name equivalence as its type checking method. Name equivalence defines two objects to be of the same type if they have the same type name. [Ref 1:p. 270] Ada also has subtypes and derived types. Subtypes alleviate some of the restrictions caused by pure name equivalence. A subtype is a subset of some base type. The subset is defined by a constraint that is applied to the base type. All operations permitted on the base type can be used with the subtype. [Ref. 1:p. 272] Derived types are derived from an existing type, either built in or user defined. The derived type is a new distinct type that has all of the capabilities of its parent, the type from which it was derived. BUT the derived type and its parent type are different types.

Figure 4 shows an example of a subtype and a derived type in Ada. Operations between the types Days_of_Week and Week_Day are permissible as the types are compatible. This provides a convenient way to express this abstraction and retain type compatibility. The Numeric_Month type inherits all of the operations available on integers but an integer type is not compatible with Numeric_Month type.

```
Type Days_of_Week is (Monday, Tuesday, Wednesday,
                      Thursday, Friday, Saturday,
                                          Sunday);

SubType Week_Day is (Monday, Tuesday, Wednesday,
               Thursday, Friday);
```

(a) Subtype Example

```
Type Numeric_Month is new Integer range 1..12;
```

(b) Derived Type Example

Figure 4. Subtype and Derived Type

Ada allows programmers to define their own types. The consistency of these types is checked both inside the current program unit and with the external units that the type interfaces with. [Ref. 7:p. 121] Two different enumerated types can have as values the same identifier. Red can be a value of the enumerated type stoplight and also be a value of the enumerated type primary. The distinction between the two reds is resolved by the context in which red is used. A variable of type primary assigned red will mean the primary red. [Ref. 1:p. 275] This allows overloading of an identifier but is still consistent with the strong typing definition.

"Red" can be used in many different contexts in the real world. There are red cars, red shirts, red lights that all use red as a adjective. Red is also a color and in this case is a noun. All of these red abstractions should be obtainable. A language should not restrict an identifier to a one time only use in an enumerated type.

Figure 5 gives an example of an incorrect procedure call that would be detected by the strong typing in Ada. If a student were using a Stack and a Tree in the same program he might make the error illustrated in Figure 5. This error could be the result of the student failing to closely check the interfaces of the ADTs being used.

```
T : Tree;
Height : Integer;
       .
       .
       .
Size (T, Height);
```

(a) Procedure Call

```
Procedure Size (S : in Stack, Depth : out Integer);
```

(b) Interface

Figure 5. Procedure Call Type Error

E. INHERITANCE

Inheritance can be very beneficial by automatically providing certain predefined or user defined capabilities and attributes. Once a programmer has solved an elementary problem he should be able to use the solution directly in solving other problems without having to restate or rewrite his first solution. Inheritance is the feature that provides the programmer this capability. Language provided capabilities for a class of objects should still be available to programmer defined subsets of that class.

Inheritance allows an abstraction that has been created to be used by other objects or ADTs that require the same or similar abstraction. Inheritance allows an abstraction to be extended to multiple ADTs at several different levels. With inheritance the programmer can take advantage of predefined abstractions when constructing new ADTs instead of having to rebuild the abstraction. Inheritance provides a programmer easy access to predefined abstractions and helps the programmer avoid duplicating abstractions.

Students can arrange ADTs that are related into a hierarchical structure. A hierarchy is a powerful abstraction itself. The "IS A" relation can be applied from the bottom of the hierarchy to the top. Students can then study ADTs with the help of the easily understood abstraction of a hierarchy.

Students are taught that certain ADTs are just special cases of other more general ADTs. For example, an AVL tree is a special case of a binary search tree, which is a special case of a binary tree, which is a special case of a tree. If a tree ADT is implemented first then all of the other trees should be easily derived. The general features of an ADT can be passed to its special cases through inheritance. The implementation of special case ADTs can be approached in the same logical manner used to describe them.

Derived types and subtypes in Ada inherit their operations from their parent type. The operations for a derived type are implicitly declared upon declaring the derived type. [Ref. 8:p. 3-11] Programmers can perform error checking more easily with subtypes by setting the range constraints of a subtype. For example, a type named numeric_month could be a subtype of the integers with the range constraint of one to 12. The automatic inheritance of integer operations by numeric_month saves the programmer from having to define these himself. Similarly the programmer can separate two abstractly different types that have the same parent and range constraint by using derived types. [Ref. 1:p. 273] For example, the type hockey_period could be derived from integers with the range constraint of one to three and the type strikes could be derived from integers with the range constraint of one to three also. Strikes and hockey_period would be incompatible types. But all of the operations allowed on integers would be available for both of these types. Inheritance of operations and attributes of the parent type by derived types and subtypes saves the programmer the problem of defining these himself.

Inheritance by derived types however is not extended to generic subprograms. [Ref. 8:p. 3-12] This prevents implementing some generic abstract data types in terms of other generic abstract data types. Specifically ADTs that can be placed in an hierarchical organization can not be implemented in terms of the parent ADT if the operations of the parent ADT are generic. If the inheritance was available then the operations of the stack and queue could be implemented in terms of a generic linked list. This implementation methodology using inheritance will be discussed further in Chapter IV.

F. EXCEPTION HANDLING

A run-time error is an <u>exception</u> to the normal course of events. [Ref. 9:p. 215] Exceptions cause a program to quit running in the expected manner. An ADT's interface is the only connection that the user has to the ADT. Run-time errors can occur in the implementation details of the ADT which are hidden from the user. The user can not fix the problem by changing these details. BUT the problem can be taken care of by an <u>exception handler</u> if the language has this feature.

Exception handling should be available to both the implementor and user of an ADT. Ada provides this capability. Ada has five pre-defined built-in exceptions that are raised when the situation dictates. [Ref. 8:p. 11-1] User defined exceptions can also be defined and raised when the programmer desires. The programmer can handle both types of exceptions in whatever manner he considers appropriate.

The ability to perform exception handling allows the author of an ADT to either tightly control all errors caused when using the ADT or loosely control them by raising exceptions and allowing the user's program to handle them. Some ADTs might require tight control while others require loose control. The key point is that either tight, loose or a combination of both can be provided when exceptions can be handled by the programmer.

When students are implementing ADTs, the point at which exceptions are handled can be dictated by the instructor. The instructor can choose where specific exceptions are to be handled. He can choose to handle some within the ADT and others to be handled outside of the ADT. Additionally, all exception handling decisions could be postponed until all other aspects of the ADT are implemented. A skeleton for handling each exception could be built inside the ADT or outside in the application, with no action being performed by using the null statement as the handling code. These null statements could then be replaced with the desired action when finalizing the ADT or application.

Exceptions should be allowed in an ADT. Most objects in the real world behave consistently until an action occurs that is an exception to the normal operating conditions. ADTs should be able to capture all of the operational characteristics of the object it is an abstraction for. Restricting an ADT's capability to only that which is considered normal fails to fully capture the object being abstracted. Students should be given the opportunity to fully capture all of the operational characteristics of an object in an ADT. Exceptions in the real world can be translated directly into exceptions in code.

<u>Exception handlers</u> are placed at the bottom of the programming unit in which an exception can occur or in which an exception is to be handled. Once an exception is raised the processing within the programming unit where it was raised is terminated. Control is passed to the exception handler at the end of the block or the end of the body of a subprogram or package. [Ref. 10:p. 315] If the exception can not be handled by that unit then the exception is propagated to another part of the program. [Ref. 9:p. 222] If the exception is never handled then the program will terminate. If it is handled then control continues past the end of the unit in which it was handled. [Ref. 10:p. 316]

Exceptions should be used in a controlled and disciplined manner. They should not be used as primary control structures. Constructs such as if-then and case statements should not be replaced with exceptions and their associated handlers. Exceptions and their handlers should augment these constructs instead of replacing them.

III. ADT INTERFACE CONSIDERATIONS

The ADT's interface is the only view a user has of the ADT. The decision a user makes about the usefulness of an ADT will be based on the information provided in the ADT interface. If a user decides to use an ADT then the correct use of the ADT is largely dependent on the interface and the user's interpretation of the information available in the interface. The interface clearly is the most important part of the ADT to the user. The following three factors influence an ADT's interface design:

1. Operations allowed on the ADT.
2. How to create the interface.
3. Interface stability relative to implementation changes.

The separation of the specification and implementation of an ADT when using Ada packages requires students to carefully consider the specification or interface of the ADT before beginning the implementation. This helps them to focus their attention on what the ADT will provide. Poor selection of ADT operations, poor documentation of the interface, and lack of implementation flexibility will negate the benefits ADTs provide.

A. DETERMINING ADT OPERATIONS

An ADT should make programming easier for its user. By providing every conceivable operation that a user may want the ADT will contain dead weight. Conversely, an ADT with a severely limited operation set may not provide all of the necessary functionality required by the user. A balance should be struck between these two extremes. The user of the ADT should be able to easily build additional operations by using the operations provided by the ADT.

The operation set of an ADT should be fully expressive. The expressiveness of an operation set can be divided into <u>expressive completeness</u> and <u>expressive richness</u>. "Expressive completeness requires that all of the computable properties on the values of a data abstraction can be expressed. Expressive richness requires in addition that all computable properties can be expressed 'simply and naturally'." [Ref. 4:p. 35] Expressive richness is the stronger of the two. [Ref. 4:p. 35]

An expressively complete data abstraction's operation set will allow all of the computable properties of the data abstraction to be expressed. BUT the complexity of expressing all of these values is not considered. An expressively rich data abstraction's operation set will allow all of the computable properties of the data abstraction to be expressed easily. [Ref. 4:p. 36] For example, the linked list ADT in Appendix A does not have a member operation. But this operation can easily be constructed by the user by returning each item in the list sequentially and checking it against the item in question.

Some languages require certain operations be performed before a declared ADT can be used. A linked list in Pascal implemented with pointers requires that the list be created by establishing the head node with a new operation. The list can not be created by simply declaring a variable of the appropriate type. This requirement is not consistent with how other data abstractions built-in to the language are used. If the programmer wants an integer, he only has to declare a variable of type integer.

The linked list ADT in Appendix A allows the user to declare a list and as a result have a list that is empty. The empty list is immediately provided because the default initialization of pointers in Ada is to null. If the implementation of the list in Appendix A had used a list instance record with the head and tail as attributes then the simple declaration of a list by the user would not result in an empty list. The list declared by the user would be a pointer to the list instance record and would be null. The attributes of the record would not be defined, even if the attributes were pointers. Assuming that the list instance record is private, the programmer has no way to obtain an empty list without invoking a create operation. The action required to create the ADT itself is therefore not implementation independent. Ideally the ADT should be created upon declaration of a variable of the ADT's type as this would be similar to what happens when a language defined type is used to declare a variable.

The operations that make up the operation set of an ADT should be restricted to those required by the user and as much as possible to those that are similar to the operations of the built-in ADTs of the programming language. This similarity should be maintained among different ADT operation sets. For example, if an ADT can be created and destroyed dynamically then all ADTs should have that possibility. The set chosen should be as rich as possible because a rich set provides maximum functionality from a small number of operations.

B. CONSTRUCTING THE INTERFACE

The interface through which the user of the ADT will gain access to the ADT must provide all of the information needed for the user to correctly use the ADT. The operations allowed must be listed with their associated parameters. The pre-conditions that must be met before an operation can occur must be provided. Similarly the result of the successful completion of the operation must be stated also. Any possible errors that may occur as a result of the operation should also be stated. The errors need not contain those that are the result of performing the operation without having met the pre-condition for the operation first. The interface should also contain any other information necessary for the successful use of the ADT. This other information could include special instructions about other actions the user needed to perform before using the ADT.

Programming languages such as Pascal, Modula-2 and Ada do not require all of the information stated above for building an ADT interface. They do require that the operations and their associated parameters be specified but the additional information is not required for a successful compilation. In Pascal and Modula-2 the additional information must be included in the form of comments embedded in the code or provided separately in a user's guide. In Pascal and Modula-2 comments are the only way for stating the additional interface information. In Ada the additional information is primarily provided via comments also. Some of the additional information in Ada can be stated with exception declarations and generic declarations. In both of these cases comments help to further specify this information. Figure 6 shows an example of the generics from the binary tree ADT found in Appendix E.

The procedure WhenTraversingDo must be defined by the user of the ADT before he can use the ADT. The generic type Item must be specified when the ADT is instantiated. The exception NOT_FOUND is declared and then a comment in the procedure Find specifies when it is raised. Some of this

```
Generic
   Type Item is Private;

with Procedure WhenTraversingDo (Content : in Item) is
   <>;
-- A user defined procedure that is compatible with this
-- should be compiled for use in the instantiation of
   this
-- generic package.
```

Figure 6. Generics to be Instantiated

information must be declared because Ada requires it but the full understanding of the information is achieved best through the use of comments.

Ideally the interface of an ADT should be completely separate from the implementation of the ADT. In Modula-2 and Ada this is achieved by the definition module and package specification respectively. Both of these are compilation units also. The separation between the interface and the implementation of the ADT helps to hide the implementation and to enforce the manipulation of the ADT through the interface only.

C. CONSISTENT INTERFACE VERSUS IMPLEMENTATION CHANGES

The ADT interface states what is done and not how it is done. This is the primary abstraction provided by an ADT. The author of the ADT is the only one to know how the ADT accomplishes what it does. The user of an ADT is only concerned with what the ADT does for him and how efficiently it is done.

The author of an ADT should create an interface that is totally independent of the implementation used. Once the initial ADT implementation is completed and the ADT is available for use, subsequent changes to the implementation should not effect the interface. If the interface is affected then the programs that have used the ADT may require changes. If it is impossible to prevent changing the interface then the changes should be kept to a minimum.

The linked list ADT in Appendix A is implemented using pointers. The linked list ADT in appendix B is implemented using an array. The interface provided to the user of both ADTs is identical with one exception. The array implementation requires that the maximum size of the list be declared when the list is instantiated. Figure 7 shows the different generic parameters that must be instantiated for the pointer and array implementations.

```
Generic
   Type Item is Private;
```

(a) Pointer Implementation

```
Generic
   Max : Integer;
   Type Item is Private;
```

(b) Array Implementation

Figure 7. Generics for Linked List Implementations

This interface change can not be prevented. By careful selection of the maximum list size, the changes required to previously written programs, which used the pointer implementation, can be restricted to the instantiation statement of the ADT.

IV. ABSTRACT DATA TYPE IMPLEMENTATION DESIGN STRATEGIES

The programmer should carefully consider what ADTs he needs and the operation set of each before implementing the first ADT. Once the ADTs have been chosen, he should check to see what ADTs are dependent on other ADTs and what ADTs are special cases of other ADTs. The ADTs that are independent and those that are parent ADTs of special case ADTs should be implemented first. This order of implementation is not required but may simplify the overall implementation effort. The other major consideration when implementing ADTs is the performance criterion of the operations. This particular aspect of the ADT can vary greatly depending on the requirements of a project but is not the focus of this thesis. For further discussion of this aspect refer to [Ref. 3].

If two ADTs A and B are to be built and B is a special case of A then the A ADT should be built first. If the language of implementation provides inheritance then B can be defined in terms of A with additional code or modifications included to take care of the specific details that make B a special case of A. If inheritance is not provided in the language then A should still be built first. B can then be defined by copying the relevant portions of code from A that would have been inherited and the additional code or modifications then made. The rationale for this ordering of implementations is simple. ADTs provide higher levels of abstraction making programming easier. The more general ADT, A, will be easier to conceptualize than the more specific, B. Likewise, the implementation of the A ADT will be easier to complete than that of B.

The ADTs found in the appendices were implemented in the same order as they appear. The linked list ADT was determined to be the atomic ADT that could be used as the parent of the stack and queue ADT. If Ada allowed inheritance of generic subprograms, then the stack and queue operations could be derived from the linked list ADT operations.

Figure 8 shows some of the operations from the linked list, queue, and stack. The two operations of the stack, Push and Pop, could be implemented with the linked list operations, Insert, Find_Item, and Delete. A Push is equivalent to an Insert at the first position of the list. A Pop is equivalent to finding the first item in the list and then deleting the item from the list.

The two operations of the queue, Enqueue and Serve, could be implemented using the Insert, Length, Delete, and Find_Item operations of the linked list. The Enqueue is equivalent to inserting at the end of the list which is the

```
    Procedure Insert (L : in out List; P : in Integer;
                                        I : in Item);

    Procedure Find_Item (L : in List; P : in Integer;
                                        I : out Item);

    Procedure Delete (L : in out List; P : in Integer;
                                        I : out Item);

    Procedure Length (L : in List; Long : out Integer);
```

(a) Linked List

```
    Procedure Push (S : in out Stack; I : in StackItem);

    Procedure Pop (S : in out Stack; I : out StackItem);
```

(b) Stack

```
    Procedure Enqueue (Q : in out Queue; I : in QueueItem);

    Procedure Serve (Q : in out Queue; I : out QueueItem);
```

(c) Queue

Figure 8. Selected ADT Operations

length of the list plus one. The Serve operation is identical to the Pop for the stack.

The level-by-level traversal algorithm used in the tree ADTs found in the appendices uses a queue. The implementations were written using a linked list. The linked list was chosen to show that the queue is only a special case of the linked list. Each of the tree ADTs contains a level-by-level traversal operation. Each of the implementations relies on a linked list ADT. The implementations could have relied on a queue ADT as well. But the point is that these implementations are dependent on another ADT. By implementing the ADTs that are used by other ADTs first, the implementations of ADTs that rely on these ADTs can be tested immediately upon completion.

The operations within a given ADT may be dependent on other operations within the same ADT. The dependent operations should be implemented after the ones they depend on. In the linked list ADT, the Length operation depends on the Empty operation. Use of ADT operations within an ADTs implementation facilitates the understanding of the implementation by taking advantage of the higher level of abstraction that the operations provide.

The Insert operation of the binary search tree and AVL-tree makes use of the exception NOT_FOUND. Figure 9 shows the code that makes up the body of the Insert from the binary search tree ADT of Appendix F. The expected result of the Find operation imbedded in the Insert operation is for the NOT_FOUND exception to be raised. This is not a classical use of an exception because the raising of the exception is the desired action. As stated earlier, exceptions should be used in a controlled and disciplined manner. The use of the NOT_FOUND exception here is acceptable because it results from taking advantage of the action the Find operation provides. No other ulterior motive exists for using the Find operation in this way. If the Find operation were not used then additional code would

```
begin
   Success := True;

   If Empty(T) then    -- Insert Item into the root node.
      TempPtr := new Node'(E,null,null);
      T.Root := TempPtr;
      T.Current := T.Root;
   Else
      Find (T,E);
      Success := False;
   end If;

   Exception
     When NOT_FOUND => --Action taken to insert new
     node.
        If Item1LessThanItem2(E,T.Current.Element) then
           T.Current.LeftChild := new Node'(E,null,
           null);
           T.Current := T.Current.LeftChild;
         Else
           T.Current.RightChild :=new Node'(E,null,
           null);
           T.Current := T.Current.RightChild;
        end If;
      When Others =>
         Raise;
end Insert;
```

Figure 9. Code of Insert Operation for Binary Search Tree

be required to move the current pointer to the appropriate node in the tree for the insertion. Code would also be required to verify that the new node being inserted was not already in the tree.

The non-classical use of an exception in the above manner allows a programmer to write code equivalent to a Boolean function with a side effect. If the Find procedure were changed to a Boolean function (True when the Item is found) then the position of the current pointer in the tree could not be altered by the Find call, as Ada will not allow an in out or out parameter in a function. The Find procedure moves the current pointer and the exception NOT_FOUND masquerades as the Boolean result. When the exception is not raised then the Boolean result is true otherwise it is false.

The tree ADTs in the appendices each contain a with clause for WhenTraversingDo. Each of the trees are generic and the data type to be embedded within a specific instance of a tree ADT is not known when writing the generic ADT. The traversal operations provided within the trees would be of no use to the user of the ADT if he could not perform some action on the content of each node in the tree. The desired action to be performed is only known by the user of the ADT. By requiring the user to define WhenTraversingDo, the author of the generic ADT can relieve himself of solving this problem.

ADTs that require ordering of data present similar problems that can be handled in a similar way. The binary search tree and AVL tree ADTs in Appendices F and G both contain a with clause for Item1LessThanItem2. This function must also be defined by the user of the ADT prior to its use. Figure 10 shows the generic declarations of the AVL tree that must be defined when the AVL tree is instantiated. The complete AVL tree can be found in Appendix G. The proper ordering of different values of the data type becomes the user's responsibility.

```
Generic
   Type Item is Private;

with Procedure WhenTraversingDo (Content : in Item) is
   <>;

with Function Item1LessThanItem2 (Item1 : in Item;
                    Item2 : in Item) Return Boolean is
   <>;
```

Figure 10. Generic Declarations of AVL Tree ADT

A programmer can save himself time and make his job easier by first studying all of the ADTs he will need for a project before implementing any of them. Ada provides an easy way to specify the what of an ADT separately from the how by using package specifications. Recognizing the interdependence among ADTs and among operations within an ADT will help a programmer to see and solve each problem only once. By doing this he can use the benefits that higher levels of abstraction provide while he creates even higher levels of abstraction.

V. CONCLUSIONS

ADTs can be taught with the primary emphasis placed on the abstraction that ADTs provide. Ada is an excellent language for teaching ADTs in this manner because it possesses the features desired for building ADTs. The language supports user defined abstractions well. The generic package capability raises the level of abstraction that one can obtain by abstracting the operations of an ADT. ADTs can be built irrespective of the data type manipulated by the ADT.

Programmers are prevented from committing common violations of ADTs by the language. Implementation details of an ADT can be completely hidden from the user of the ADT through the use of packages, private types and limited private types. Run time exceptions can be handled by the language. These exceptions can be handled within the ADT or by the user of the ADT.

The inheritance capability provided by Ada gives the programmer an easy way to define new types from parent types and retain the operations applicable to the parent type for the new type. The major failing of the inheritance capability is the lack of generic subprogram inheritance. This prevents defining one generic ADT in terms of another generic ADT.

Ada is a very complex language and is more difficult to learn than Pascal or Modula-2. This complexity is the major detractor from using the language for teaching ADTs. A programmer with previous Ada programming experience can easily write ADTs in Ada. A novice programmer with no Ada experience, however, will find the language quite difficult.

Ada provides many features that support abstraction and the writing of robust ADTs. Generics, packages, separate compilation, limited inheritance and exceptions all contribute to the security and abstraction desired of an ADT. The complexity of the language and the absence of complete inheritance among all ADTs weaken Ada as a language for ADTs. But the strengths of Ada are far greater than these two weaknesses. In fact, if Ada were not as complex, it would probably not have all of the strengths for ADTs that it does.

Programmers should try to write generic ADTs instead of specific ADTs. The advantage of generics over specifics is the greater useability of an ADT. Packages should be used as the programming unit because of the interface/ implementation separation provided by the specification and body portions of the package.

When several ADTs are to be built and used together, the programmer should specify the operation set of each ADT before implementing the first ADT. The operation set chosen for each ADT should be expressively rich. Programmer defined exceptions and where these exceptions will be handled should also be specified before the first ADT is implemented.

Programmers should use the inheritance Ada provides. Subtypes and derived types coupled with Ada's strong typing can ensure better control of logically different data types. Similarly, programmers should use private and limited private types within packages to ensure the security of the underlying hidden data structure.

Programmers with no Ada knowledge should be taught Ada first or use another language to learn about ADTs. If time permits Ada should be learned and then ADTs pursued. The benefits of Ada for learning ADTs outweigh the difficulty of learning Ada. BUT to write good ADTs in Ada, the programmer must know how to use the many features of Ada and he must use these wisely.

APPENDIX A

LINKED LIST POINTER IMPLEMENTATION

All programs have been compiled using the Meridian AdaVantage Compiler (version 2.1) on an IBM XT.

```
Generic
  Type Item is Private;

Package Generic_List is
  Type List is Limited Private;

BEYOND_END : Exception;
NOT_FOUND : Exception;
INSERT_BEYOND : Exception;
DELETE_OUT_OF_RANGE : Exception;

Procedure Clear (L : in out List);
-- pre - None.
-- post - L-pre exists as an empty list.

Function Full(L : in List) Return Boolean;
-- pre - None.
-- post - True if the list L can not have more items added,
--        otherwise False.

Function Empty(L : in List) Return Boolean;
-- pre - None.
-- post - True if list L has no items in it, otherwise False.

Procedure Insert (L : in out List; P : in Integer;
                                  I : in Item);
-- pre - The size of L has not reached its maximum.
-- post - L includes item I in the Pth place
-- exceptions raised - INSERT_BEYOND
--                     if P > (Length of list + 1)
```

```
Procedure Delete (L : in out List; P : in Integer;
                                    I : out Item);
-- pre - L is not empty.
-- post - I was the Pth item of the List.
--       L no longer contains I.
-- exceptions raised -DELETE_OUT_OF_RANGE if P > Length of L.

Procedure Length (L : in List; Long : out Integer);
-- pre - L exists.
-- post - Long is equal to the number of items in L.


Procedure Find_Item (L : in List; P : in Integer;
                                    I : out Item);
-- pre - L is not empty.
-- post - I is the Pth item of L.  L is unchanged.
-- exception raised - BEYOND_END if P > Length of List.

Procedure Find_Pos (L : in List; P : out Integer;
                                    I : in Item);
-- pre - L is not empty.
-- post - P is the position of I in  L.
-- exception raised - NOT_FOUND if I is not found in List.

Private
  Type Node;
  Type List is Access Node;

end Generic_List;


with Unchecked_Deallocation;
Package body Generic_List is

Type Node is
  Record
    Element : Item;
    Next : List;
  end Record;


Procedure Return_Node is new Unchecked_Deallocation
                                    (Node, List);
```

```
Procedure Clear (L : in out List) is
-- post - L-pre exists as an empty list.

Temp_Ptr : List;

begin
  If not Empty(L) then
    While (L.Next /= null)   -- Reclaims each node in list
    Loop                      -- except last one.
      Temp_Ptr := L;
      L := L.Next;
      Return_Node (Temp_Ptr);
    end Loop;

    Return_Node (L);        -- Reclaims last node in list.
  end If;
end Clear;



Function FULL(L : in List) Return Boolean is
-- post - True if the list L can not have more items added,
--       otherwise False.

Temp_Ptr : List;

 begin

   Temp_Ptr := new Node;        -- Generates new pointer
   Return_Node(Temp_Ptr);       -- Returns pointer to memory
   Return (false);

   Exception
     when STORAGE_ERROR =>
       Return (true);        -- Out of memory.
     when others =>
       Raise;
end FULL;


Function Empty(L : in List) Return Boolean is
-- post - True if list L has no items in it, otherwise False.
```

```
begin
  Return (L = null);
end Empty;


Procedure Insert (L : in out List; P : in Integer;
                                      I : in Item) is
-- pre - The size of L has not reached its maximum.
-- post - L includes item I in the Pth place
-- exceptions raised - INSERT_BEYOND
--                  if P > (Length of list + 1).

Before_New : List;
Temp_Ptr : List;
New_Ptr : List;
Previous : Integer;
NumItems : Integer;

begin

  New_Ptr := new Node'(I,null);
                   -- Establishes node to be inserted.

  Previous := P;
  Temp_Ptr := L;

  If P = 1 then              -- Inserting at front of list.
    New_Ptr.Next := L;
    L := New_Ptr;
  Else
    If Temp_Ptr = null then
      Raise (INSERT_BEYOND);
    end If;

    While (Previous /= 1)  --Before_New will be pointing to
    Loop                   -- item in list that will precede
      Previous := Previous - 1;     -- item being inserted.
      Before_New := Temp_Ptr;
      Temp_Ptr := Temp_Ptr.Next;  --Temp_Ptr points to
                             --item that succeeds new
                             --item in list.
      If Temp_Ptr = null then
        Exit;
      end If;
    end Loop;

    If Previous = 1 then
      Before_New.Next := New_Ptr;
      New_Ptr.Next := Temp_Ptr;
    Else
      raise (INSERT_BEYOND);
    end If;
  end If;

end Insert;


Procedure Delete (L : in out List; P : in Integer;
                                    I : out Item) is
-- pre - L is not empty.
-- post - I was the Pth item of the List.
--       L no longer contains I.
-- exceptions raised - DELETE_OUT_OF_RANGE
--                   if P > the length of L.

Temp_Ptr : List;
Node_Before : List;
Count : Integer;
NumItems : Integer;

begin
  Temp_Ptr := L;

  If P = 1 then          -- Deletion if first item in list.
    I := Temp_Ptr.Element;
    L := Temp_Ptr.Next;
    Return_Node(Temp_Ptr);

  Else

    For Count in 1..(P-1)  -- Node_Before will point to
    Loop                   -- item in list before the one
                                 -- to be deleted.
      Node_Before := Temp_Ptr;
      If Temp_Ptr = null then
         Raise (DELETE_OUT_OF_RANGE);
                   -- Can't delete beyond end of list.
       end If;
       Temp_Ptr := Temp_Ptr.Next;  -- Temp_Ptr will point
    end Loop;                   --to item to be deleted.

    If Temp_Ptr = null then
      Raise (DELETE_OUT_OF_RANGE);
                   -- Can't delete beyond end of list.
    end If;

    Node_Before.Next := Temp_Ptr.Next;
    I := Temp_Ptr.Element; --Content of node being deleted.
    Return_Node(Temp_Ptr); --Reclaims pointer to node
                                  -- deleted.
  end If;
end Delete;


Procedure Length (L : in List; Long : out Integer) is
-- pre - L exists.
-- post - Long is equal to the number of items in L.

Temp_Ptr : List;
Count : Integer;

begin
  If Empty(L) then
    Long := 0;               -- Returns length.
  Else
    Temp_Ptr := L;
    Count := 1;

    While (Temp_Ptr.Next /= null)  -- Traverse list
    Loop                    -- incrementing count
                         -- at each step.
      Count := Count + 1;
      Temp_Ptr := Temp_Ptr.Next;
    end Loop;

    Long := Count;             -- Returns length.
  end If;
end Length;


Procedure Find_Item (L : in List; P : in Integer;
                                   I : out Item) is
-- pre - L is not empty.
-- post - I is the Pth item of L.  L is unchanged.
-- exception raised - BEYOND_END if P > length of the list.

NumItems : Integer;
Temp_Ptr : List;
Count : Integer;

begin
  Temp_Ptr := L;

  For Count in 1..(P-1)  -- Traverse list to the Pth item.
  Loop
    Temp_Ptr := Temp_Ptr.Next;
    If Temp_Ptr = null then
      raise (BEYOND_END);  -- Can't find Pth item when
    end If;              -- list length is less than P.
  end Loop;

  I := Temp_Ptr.Element;     -- Returns the Pth item.
end Find_Item;


Procedure Find_Pos (L : in List; P : out Integer;
                                   I : in Item) is
-- pre - L is not empty.
-- post - P is the position of I in  L.
-- exception raised - NOT_FOUND if I is not found in list.

Temp_Ptr : List;
New_Ptr : List;
Previous : List;
Count : Integer;

begin
  Temp_Ptr := L;
  If Temp_Ptr.Element = I then  -- First item in the list.
     P := 1;
   Else
     Count := 1;

     While (Temp_Ptr.Element /= I) and
       (Temp_Ptr.Next /= null)  -- Traverse list until
     Loop                        -- found or end of list.
       Temp_Ptr := Temp_Ptr.Next;
       Count := Count + 1;       -- Count each node checked.
     end Loop;

     If Temp_Ptr.Element /= I then
       raise NOT_FOUND;  --Item desired is not in the list.
     Else
       P := Count;     -- Item located in the Pth position.
     end If;
  end If;
end Find_Pos;

end Generic_List;
```

Instantiation of Generic_List

```
with ada_io;
use ada_io;
with Generic_List;

Procedure listtest is
Package IntList is new Generic_List(Item => Integer);

S : IntList.List;
Int1 : INTEGER;
Int2 : INTEGER;
Bool1 : BOOLEAN;

begin
  Loop
    new_line;
```

```
put ("1. Full?");
new_line;
put ("2. Empty?");
new_line;
put ("3. Clear.");
new_line;
put ("4. Insert.");
new_line;
put ("5. Delete.");
new_line;
put ("6. Length.");
new_line;
put ("7. Find Item.");
new_line;
put ("8. Find Position.");
new_line;
new_line;
put ("Enter number of action you desire. ");
get (Int1);

Case Int1 is
  when 1 =>
    new_line;
    put("Checking if list is full.");
    If IntList.Full(S) then
      put(" Yes it is full.");
    Else
      put(" No it is not full.");
    end If;

  when 2 =>
    new_line;
    put ("Checking if list is empty.");
    If IntList.Empty(S) then
      put(" Yes it is empty.");
    Else
      put(" No it is not empty.");
    end If;

  when 3 =>
    new_line;
    put("Clearing the list.");
    IntList.Clear(S);
```

```
   put(" Finished clearing the list.");

 when 4 =>
   new_line;
   put("Enter integer you want to put in list. ");
   get(Int2);
   new_line;
   put("Enter position in list where ");
   put(Int2);
   put(" should be inserted. ");
   get(Int1);
   IntList.Insert(S,Int1,Int2);
   new_line;
   put("Finished with insertion.");

 when 5 =>
   new_line;
   put("Enter position you want deleted from the");
   put(" list. ");
   get(Int1);
   new_line;
   IntList.Delete(S,Int1,Int2);
   put(Int2);
   put(" was deleted from the list and was in");
   put(" position ");
   put(Int1);
   put(".");
   new_line;

 when 6 =>
   new_line;
   IntList.Length(S,Int1);
   put("The length of the list is ");
   put(Int1);
   put(".");
   new_line;

 when 7 =>
   new_line;
   put("Enter the position in the list you want");
   put(" the content of. ");
   get(Int1);
   new_line;
```

```
        IntList.Find_Item(S,Int1,Int2);
        put(Int2);
        put(" was in the ");
        put(Int1);
        put(" position.");

      when 8 =>
        new_line;
        put("Enter integer you need the position of");
        put(" in the list. ");
        get(Int2);
        new_line;
        IntList.Find_Pos(S,Int1,Int2);
        put(Int2);
        put(" was in the ");
        put(Int1);
        put(" position of the list.");

      when others =>
        EXIT;
    end Case;
  end Loop;


  Exception
    when IntList.BEYOND_END =>
      put ("Trying to get something from beyond end of");
      put (" the list.");
    when IntList.NOT_FOUND =>
      put ("What you were looking for in the list isn't");
      put (" there.");
    when IntList.DELETE_OUT_OF_RANGE =>
      put ("You can not delete beyond the end of the");
      put (" list.");
    when IntList.INSERT_BEYOND =>
      put ("You can not insert beyond the end of a"):
      put (" list.");
    when others =>
      Raise;
end ListTest;
```

APPENDIX B

LINKED LIST ARRAY IMPLEMENTATION

```
Generic
  Max : Integer;
  Type Item is Private;

Package Array_List is
  Type List is Limited Private;


BEYOND_END : Exception;
NOT_FOUND : Exception;
INSERT_BEYOND : Exception;
DELETE_OUT_OF_RANGE : Exception;

Procedure Clear (L : in out List);
-- pre - None.
-- post - L-pre exists as an empty list.

Function Full(L : in List) return Boolean;
-- pre - None.
-- post - True if the list L can not have more items added,
--       otherwise False.

Function Empty(L : in List) return Boolean;
-- pre - None.
-- post - True if list L has no items in it, otherwise False.

Procedure Insert (L : in out List; P : in Integer;
                               I : in Item);
-- pre - The size of L has not reached its maximum.
-- post - L includes item I in the Pth place
-- Exception raised - INSERT_BEYOND
--                    if P > (Length of list + 1).

Procedure Delete (L : in out List; P : in Integer;
                               I : out Item);
```

```
-- pre - L is not empty.
-- post - I was the Pth item of the List.
--       L no longer contains I.
-- exceptions rasied - DELETE_OUT_OF_RANGE
--                              if P > the length of L.


Procedure Length (L : in List; Long : out Integer);
-- pre - L exists.
-- post - Long is equal to the number of items in L.


Procedure Find_Item (L : in List; P : in Integer;
                                    I : out Item);
-- pre - L is not empty.
-- post - I is the Pth item of L.  L is unchanged.
-- Exception raised - BEYOND_END if P > Length of list.


Procedure Find_Pos (L : in List; P : out Integer;
                                    I : in Item);
-- pre - L is not empty.
-- post - P is the position of I in  L.
-- Exception raised - NOT_FOUND
--                        if I is not found in the list.

Private
  Type ListArray is array (1..Max) of Item;
  Type List is
    Record
      ListLength : Integer:= 0;
      Lst : ListArray;
    end Record;

end Array_List;


Package body Array_List is

Procedure Clear (L : in out List) is
-- post - L-pre exists as an empty list.
```

```
begin
  L.ListLength := 0;   -- Satisfies Empty list condition.
end Clear;


Function Full(L : in List) return Boolean is
-- post - True if the list L can not have more items added,
--       otherwise False.

begin
  Return (L.ListLength = Max);
end Full;


Function Empty(L : in List) return Boolean is
-- post - True if list L has no items in it, otherwise False.

begin
  Return (L.ListLength = 0);
end Empty;


Procedure Insert (L : in out List; P : in Integer;
                                I : in Item) is
-- pre - The size of L has not reached its maximum.
-- post - L includes item I in the Pth place
-- Exceptions raised - INSERT_BEYOND
--                           if P is > L's length + 1.

NumItems : Integer;
NewItem : Item;
TempItem : Item;

begin
  Length(L,NumItems);
  NewItem := I;

  If P > (NumItems + 1) then
    Raise (INSERT_BEYOND);  -- Can't insert beyond end of
                            --list.
  Else
```

```
    If P = (NumItems + 1) then  -- Inserting at end of the
      L.Lst(P) := I;                       -- list.
      L.ListLength := L.ListLength + 1;

    Else

      For Index in P..(NumItems + 1)
      Loop        -- Insert new item & move everyone else
        TempItem := L.Lst(Index);   -- down one position.
        L.Lst(Index) := NewItem;
        NewItem := TempItem;
      end Loop;

      L.ListLength := L.ListLength + 1;

    end If;
  end If;
end Insert;


Procedure Delete (L : in out List; P : in Integer;
                                    I : out Item) is
-- pre - L is not empty.
-- post - I was the Pth item of the List.
-- L no longer contains I.
-- exceptions raised - DELETE_OUT_OF_RANGE
--                          if P > the length of list.

NumItems : Integer;

begin
  Length(L,NumItems);
  If P > NumItems then
    Raise (DELETE_OUT_OF_RANGE);  -- Can't delete beyond
                               -- end of the list.
  Else

    I := L.Lst(P);
    L.ListLength := L.ListLength - 1;
    If P = NumItems then  --Deleting last item in the list.
      null;            -- No need to shift down the list.

    Else
```

```
      For Index in P..(NumItems - 1) -- Moves items after
      Loop                    -- P up 1 position.
        L.Lst(Index) := L.Lst((Index + 1));
      end Loop;

    end If;
  end If;
end Delete;


Procedure Length (L : in List; Long : out Integer) is
-- pre - L exists.
-- post - Long is equal to the number of items in L.

begin
  Long := L.ListLength;
end Length;


Procedure Find_Item (L : in List; P : in Integer;
                              I : out Item) is
-- pre - L is not empty.
-- post - I is the Pth item of L.  L is unchanged.
-- Exception raised - BEYOND_END if P > length of the List.

NumItems : Integer;

begin
  Length(L,NumItems);

  If P > NumItems then
    Raise (BEYOND_END);  -- Can't find item beyond end of
  Else                        -- list.
    I := L.Lst(P);      -- Returns desired item.
  end If;
end Find_Item;



Procedure Find_Pos (L : in List; P : out Integer;
                              I : in Item) is
-- pre - L is not empty.
-- post - P is the position of I in  L.
-- Exception raised - NOT_FOUND
--                        if I is not found in the List.

NumItems : Integer;
Found : Boolean;
Index : Integer;

begin
  Length(L,NumItems);
  Index := 1;
  Found := false;

  While (Index <= NumItems) and not (Found)
  Loop                  -- Traverse list looking for match.
    If L.Lst(Index) = I then
      Found := true;
      P := Index;
    Else
      Index := Index + 1;
    end If;
  end Loop;

  If not (Found) then  -- List traversed and item not found.
    Raise (NOT_FOUND);
  end If;
end Find_Pos;
end Array_List;
```

Instantiation of Array_List

```
with ada_io;
use ada_io;
with Array_list;

Procedure lasttest is
Package LSTINT is new Array_list(MAX => 60, ITEM => Integer);

S : LSTINT.List;
Int1 : INTEGER;
Int2 : INTEGER;
```

```
Bool1 : BOOLEAN;

begin
  Loop
    new_line;
    put ("1. Full?");
    new_line;
    put ("2. Empty?");
    new_line;
    put ("3. Clear.");
    new_line;
    put ("4. Insert.");
    new_line;
    put ("5. Delete.");
    new_line;
    put ("6. Length.");
    new_line;
    put ("7. Find Item.");
    new_line;
    put ("8. Find Position.");
    new_line;
    new_line;
    put ("Enter number of action you desire. ");
    get (Int1);

    Case Int1 is
      when 1 =>
        new_line;
        put("Checking if list is full.");
        If LSTINT.Full(S) then
          put(" Yes it is full.");
        Else
          put(" No it is not full.");
        end If;

      when 2 =>
        new_line;
        put ("Checking if list is empty.");
        If LSTINT.Empty(S) then
          put(" Yes it is empty.");
        Else
          put(" No it is not empty.");
        end If;
```

```
when 3 =>
  new_line;
  put("Clearing the list.");
  LSTINT.Clear(S);
  put(" Finished clearing the list.");

when 4 =>
  new_line;
  put("Enter integer you want to put in list. ");
  get(Int2);
  new_line;
  put("Enter position in list where ");
  put(Int2);
  put(" should be inserted. ");
  get(Int1);
  LSTINT.Insert(S,Int1,Int2);
  new_line;
  put("Finished with insertion.");

when 5 =>
  new_line;
  put("Enter position you want deleted from the");
  put(" list. ");
  get(Int1);
  new_line;
  LSTINT.Delete(S,Int1,Int2);
  put(Int2);
  put(" was deleted from the list and was in");
  put (" position ");
  put(Int1);
  put(".");
  new_line;

when 6 =>
  new_line;
  LSTINT.Length(S,Int1);
  put("The length of the list is ");
  put(Int1);
  put(".");
  new_line;
```

```
    when 7 =>
      new_line;
      put("Enter the position in the list you want");
      put(" the content of. ");
      get(Int1);
      new_line;
      LSTINT.Find_Item(S,Int1,Int2);
      put(Int2);
      put(" was in the ");
      put(Int1);
      put(" position.");

    when 8 =>
      new_line;
      put("Enter integer you need the position of in");
      put(" the list. ");
      get(Int2);
      new_line;
      LSTINT.Find_Pos(S,Int1,Int2);
      put(Int2);
      put(" was in the ");
      put(Int1);
      put(" position of the list.");

    when others =>
      EXIT;
  end Case;
end Loop;


Exception
  when LSTINT.BEYOND_END =>
    put ("Trying to get something from beyond end of");
    put (" the list.");
  when LSTINT.NOT_FOUND =>
    put ("What you were looking for in the list isn't");
    put (" there.");
    put ("  You can not delete from an empty list.");
  when LSTINT.DELETE_OUT_OF_RANGE =>
    put ("You can not delete beyond the end of the");
    put (" list.");
  when LSTINT.INSERT_BEYOND =>
    put ("You can not insert beyond the end of a");
```

```
        put (" list.");
      when others =>
        Raise;
end lasttest;
```

APPENDIX C

STACK

```
Generic
  Type StackItem is Private;

Package Gen_Stack is
  Type Stack is Limited Private;


Procedure Clear (S : in out Stack);
-- pre - None.
-- post - S is an empty stack.

Function Full(S : in Stack) Return Boolean;
-- pre - None.
-- post - True if stack S can not have more items added,
--        otherwise False.

Function Empty(S : in Stack) Return Boolean;
-- pre - None.
-- post -True if stack S has no items in it, otherwise False.

Procedure Push (S : in out Stack; I : in StackItem);
-- pre - The size of S has not reached its maximum.
-- post - S has item I on top of it.

Procedure Pop (S : in out Stack; I : out StackItem);
-- pre - S is not empty.
-- post - Top of S-pre is assigned to I and S no longer
--        contains I.

Procedure Size (S : in Stack; Depth : out Integer);
-- pre - S exists.
-- post - Depth is equal to the number of items in S.

Procedure Top (S : in Stack; I : out StackItem);
-- pre - S is not empty.
-- post - I is the top item of S.  S is unchanged.


Private
  Type Plate;
  Type Stack is Access Plate;

end Gen_Stack;



with Unchecked_Deallocation;
Package body Gen_Stack is

Type Plate is
  Record
    Element : StackItem;
    Next : Stack;
  end Record;

Procedure Return_Plate is
            new Unchecked_Deallocation (Plate, Stack);


Procedure Clear (S : in out Stack) is
-- post - S is an empty stack.

Temp_Ptr : Stack;

begin
  If not Empty(S) then
    While (S.Next /= null)  -- Reclaims each node in Stack
    Loop                     -- except last one.
      Temp_Ptr := S;
      S := S.Next;
      Return_Plate (Temp_Ptr);
    end Loop;

    Return_Plate (S);     -- Reclaims last node in Stack.
  end If;
end Clear;
```

```
Function FULL(S : in Stack) Return Boolean is
-- post - True if the stack S can not have more items pushed,
--       otherwise False.

Temp_Ptr : Stack;

begin
  Temp_Ptr := new Plate;      -- Generates new pointer.
  Return_Plate(Temp_Ptr);     -- Returns pointer to memory.
  Return (false);

Exception
  when STORAGE_ERROR =>
    Return (True);              -- Out of memory.
  when others  =>
    Raise;
end FULL;


Function Empty(S : in Stack) Return Boolean is
-- post -True if stack S has no items in it, otherwise False.

begin
  Return (S = null);
end Empty;


Procedure Push (S : in out Stack; I : in StackItem) is
-- pre - The size of S has not reached its maximum.
-- post - S includes item I on the top of the stack.

New_Ptr : Stack;

begin
  New_Ptr := new Plate'(I,S);  -- Creates new node on top of
                                -- stack.

  S := New_Ptr;        -- Assigns S to new top of the stack.
end Push;


Procedure Pop (S : in out Stack; I : out StackItem) is
-- pre - S is not empty.
-- post - Top of S-pre is assigned to I and S no longer
--       contains I.

Temp_Ptr : Stack;

begin
  Temp_Ptr := S;

  I := S.Element;        -- Retrieves Item on top of stack.

  S := S.Next;        -- Assigns S to new top of the stack.

  Return_Plate(Temp_Ptr); --Reclaims pointer that was the
                          -- top of the stack.
end Pop;


Procedure Size (S : in Stack; Depth : out Integer) is
-- pre - S exists.
-- post - Depth is equal to the number of items in S.

Temp_Ptr : Stack;
Count : Integer;

begin
  If Empty(S) then
    Depth := 0;                -- Returns Depth.
  Else
    Temp_Ptr := S;
    Count := 1;

    While (Temp_Ptr.Next /= null)  -- Count items in the
    Loop                           -- stack.
      Count := Count + 1;
      Temp_Ptr := Temp_Ptr.Next;
    end Loop;

    Depth := Count;            -- Returns Depth.
  end If;
end Size;
```

```
Procedure Top (S : in Stack; I : out StackItem) is
-- pre - S is not empty.
-- post - I is the top item of S.  S is unchanged.

begin
  I := S.Element;  -- Copies into I content of Top of the
                                           -- stack.
end Top;
end Gen_Stack;
```

Instantiation of Gen_Stack

```
with ada_io;
use ada_io;
with Gen_Stack;
Procedure stcktest is
Package Int_Stck is new Gen_Stack(StackItem => Integer);

S : Int_Stck.Stack;
Int1 : INTEGER;
Int2 : INTEGER;
Bool1 : BOOLEAN;

begin
  Loop
    new_line;
    put ("1. Clear");
    new_line;
    put ("2. Empty?");
    new_line;
    put ("3. Full?");
    new_line;
    put ("4. Push.");
    new_line;
    put ("5. Pop.");
    new_line;
    put ("6. Size.");
    new_line;
    put ("7. Top.");
    new_line;
```

```
new_line;
put ("Enter number of action you desire. ");
get (Int1);

Case Int1 is
  when 1 =>
    new_line;
    put("Clearing the stack.");
    Int_Stck.Clear(S);
    new_line;
    put(" Stack is cleared.");

  when 2 =>
    new_line;
    put ("Checking if stack is empty.");
    If Int_Stck.Empty(S) then
      put(" Yes it is empty.");
    Else
      put(" No it is not empty.");
    end If;

  when 3 =>
    new_line;
    put("Checking if the stack is full.");
    new_line;
    If Int_Stck.Full(S) then
      put("Stack is full.");
    Else
      put ("Stack is not full.");
    end If;

  when 4 =>
    new_line;
    put("Enter integer you want to push on stack. ");
    get(Int2);
    new_line;
    Int_Stck.Push(S,Int2);
    new_line;
    put("Finished with Push.");

  when 5 =>
    new_line;
    Int_Stck.Pop(S,Int2);
```

```
          put(Int2);
          put(" was just Popped from the stack.");

        when 6 =>
          new_line;
          Int_Stck.Size(S,Int1);
          put("The depth of the stack is ");
          put(Int1);
          put(".");

        when 7 =>
          new_line;
          Int_Stck.Top(S,Int2);
          put(Int2);
          put(" is on the Top of the stack.");

        when others =>
          EXIT;
      end Case;
    end Loop;

  end StckTest;
```

APPENDIX D

QUEUE

```
Generic
  Type QueueItem is Private;

Package Gen_Queue is
  Type Queue is Limited Private;


Procedure Clear (Q : in out Queue);
-- pre - None.
-- post - Q-pre exists as an empty queue.

Function Full(Q : in Queue) Return Boolean;
-- pre - None.
-- post - True if queue Q can not have more items added,
--       otherwise False.

Function Empty(Q : in Queue) Return Boolean;
-- pre - None.
-- post -True if queue Q has no items in it, otherwise False.

Procedure Enqueue (Q : in out Queue; I : in QueueItem);
-- pre - The size of Q has not reached its maximum.
-- post - Q has item I at end of it.

Procedure Serve (Q : in out Queue; I : out QueueItem);
-- pre - Q is not empty.
-- post - I was at front of the Queue Q.  Q no longer
--       contains I.

Procedure Size (Q : in Queue; Length : out Integer);
-- pre - Q exists.
-- post - Length is equal to the number of items in Q.

Procedure Front (Q : in Queue; I : out QueueItem);
-- pre - Q is not empty.
-- post - I is the first item of Q.  Q is unchanged.


Private
  Type Q_Item;
  Type Queue is Access Q_Item;

end Gen_Queue;


with Unchecked_Deallocation;
Package body Gen_Queue is

Type Q_Item is
  Record
    Element : QueueItem;
    Next : Queue;
  end Record;


Procedure Return_Q_Item is
            new Unchecked_Deallocation (Q_Item, Queue);


Procedure Clear (Q : in out Queue) is
-- post - Q-pre exists as an empty queue.

Temp_Ptr : Queue;

begin
  If not Empty(Q) then
    While (Q.Next /= null) --Reclaims each node in Queue
    Loop                   -- except last one.
      Temp_Ptr := Q;
      Q := Q.Next;
      Return_Q_Item (Temp_Ptr);
    end Loop;

    Return_Q_Item (Q);   -- Reclaims last node in Queue
  end If;
end Clear;
```

```
Function FULL(Q : in Queue) Return Boolean is
-- post - True if the queue Q can not have more items added,
--       otherwise False.

Temp_Ptr : Queue;

begin
  Temp_Ptr := new Q_Item;        -- Generates new pointer.
  Return_Q_Item(Temp_Ptr);   -- Returns pointer to memory.
  Return (false);

  Exception
    when STORAGE_ERROR =>
      Return (True);              -- Out of memory.
    when others  =>
      Raise;
end FULL;


Function Empty(Q : in Queue) Return Boolean is
-- post -True if queue Q has no items in it, otherwise False.

begin
  Return (Q = null);
end Empty;


Procedure Enqueue (Q : in out Queue; I : in QueueItem) is
-- pre - The size of Q has not reached its maximum.
-- post - Q includes item I at the end of the queue.

New_Ptr : Queue;
End_Ptr : Queue;

begin
  New_Ptr := new Q_Item'(I,null);  -- Creates new node to go
                                   -- in queue.
  If Empty(Q) then
    Q := New_Ptr;
  Else
    End_Ptr := Q;         - Initializes to front of queue.

    While (End_Ptr.Next /= null)  -- Travels length of
    Loop                    -- queue to end.
      End_Ptr := End_Ptr.Next;
    end Loop;

    End_Ptr.Next := New_Ptr; -- Adds new QueueItem to end
  end If;                            -- of Queue.
end Enqueue;


Procedure Serve (Q : in out Queue; I : out QueueItem) is
-- pre - Q is not empty.
-- post - I was at front of the Queue Q.
--       Q no longer contains I.

Temp_Ptr : Queue;

begin
  Temp_Ptr := Q;

  I := Temp_Ptr.Element; -- Retrieves Item on top of queue.

  Q := Temp_Ptr.Next;  --Assigns Q to new top of the queue.

  Return_Q_Item(Temp_Ptr); -- Reclaims pointer that was the
                           -- top of the queue.
end Serve;


Procedure Size (Q : in Queue; Length : out Integer) is
-- pre - Q exists.
-- post - Length is equal to the number of items in Q.

Temp_Ptr : Queue;
Count : Integer;

begin
  If Empty(Q) then
    Length := 0;              -- Returns Length.
  Else
    Temp_Ptr := Q;
    Count := 1;

    While (Temp_Ptr.Next /= null)  -- Count items in the
    Loop                              -- queue.
      Count := Count + 1;
      Temp_Ptr := Temp_Ptr.Next;
    end Loop;

    Length := Count;              -- Returns Length.
  end If;
end Size;


Procedure Front (Q : in Queue; I : out QueueItem) is
-- pre - Q is not empty.
-- post - I is the top item of Q.  Q is unchanged.

begin
  I := Q.Element;  -- Copies into I content of Front of
                   -- the queue.
end Front;
end Gen_Queue;
```

Instantiation of Gen_Queue

```
with ada_io;
use ada_io;
with Gen_Queue;
Procedure quetest is
Package Int_Queue is new Gen_Queue(QueueItem => Integer);

Q : Int_Queue.Queue;
Int1 : INTEGER;
Int2 : INTEGER;
Bool1 : BOOLEAN;

begin
  Loop
    new_line;
    put ("1. Clear");
    new_line;
    put ("2. Empty?");
    new_line;
```

```
put ("3. Full?");
new_line;
put ("4. Enqueue.");
new_line;
put ("5. Serve.");
new_line;
put ("6. Size.");
new_line;
put ("7. Front.");
new_line;
new_line;
put ("Enter number of action you desire. ");
get (Int1);

Case Int1 is
  when 1 =>
    new_line;
    put("Clearing the queue.");
    Int_Queue.Clear(Q);
    new_line;
    put("Queue is cleared.");

  when 2 =>
    new_line;
    put ("Checking if queue is empty.");
    If Int_Queue.Empty(Q) then
      put(" Yes it is empty.");
    Else
      put(" No it is not empty.");
    end If;

  when 3 =>
    new_line;
    put("Checking if the queue is full.");
    new_line;
    If Int_Queue.Full(Q) then
      put("Queue is full.");
    Else
      put ("Queue is not full.");
    end If;

  when 4 =>
    new_line;
```

```
          put("Enter integer you want to place in the queue. ");
          get(Int2);
          new_line;
          Int_Queue.Enqueue(Q,Int2);
          new_line;
          put("Finished with enqueue.");

        when 5 =>
          new_line;
          Int_Queue.Serve(Q,Int2);
          put(Int2);
          put(" was just Served from the queue.");

        when 6 =>
          new_line;
          Int_Queue.Size(Q,Int1);
          put("The size of the queue is ");
          put(Int1);
          put(".");

        when 7 =>
          new_line;
          Int_Queue.Front(Q,Int2);
          put(Int2);
          put(" is at the Front of the queue.");

        when others =>
          EXIT;
      end Case;
    end Loop;

  end QueTest;
```

APPENDIX E

BINARY TREE

```
Generic
  Type Item is Private;


with Procedure WhenTraversingDo (Content : in Item) is <>;
-- A user defined procedure that is compatible with this
-- should be compiled for use in the instantiation of this
-- generic package.

Package Gen_BinaryTree is
  Type BinaryTree is Limited Private;

NOT_FOUND : Exception;


Procedure Clear (T : in out BinaryTree);
-- pre - T exists.
-- post - T-pre exists as an empty tree.

Procedure Destroy (T : in out BinaryTree);
-- pre - T exists.
-- post - T-pre no longer exists.

Function Full(T : in BinaryTree) Return Boolean;
-- pre - T exists.
-- post - True if the tree T can not have more items added,
-- otherwise False.

Function Empty(T : in BinaryTree) Return Boolean;
-- pre - T exists.
-- post - True if tree T has no items in it, otherwise False.

Procedure Create(T : in out BinaryTree);
-- pre - None.
-- post - T exists and is empty.
```

```
Procedure Insert (T : in out BinaryTree; E : in Item;
            C : in Character; Success : in out Boolean);
-- pre - T exists. The size of T has not reached its maximum.
-- comment - Character 'L' or 'l' should be parameter for
--                                   left child.
--        Character 'R' or 'r' should be parameter for
--                                   right child.
-- post - If pre-T was empty then E is the content of the
--      Root node of T and Success is true.
--      Otherwise pre-Current location has a new C child
--      that contains E if Success is true and Current
--      location is the new child.
--      Success is false if C child existed prior to
--      insertion try.  Current location remains same as
--      pre if Success is false.

Procedure DeleteSub (T : in out BinaryTree);
-- pre - T is not empty.
-- post - SubTree rooted at pre-Current location is deleted.
--      If pre-Current is root then tree no longer exists.

Procedure Update (T : in out BinaryTree; E : in Item);
-- pre - T is not empty.
-- post - E is the new content of Current location.

Procedure Retrieve (T : in BinaryTree; E : in out Item);
-- pre - T is not empty.
-- post - E is the content of pre-Current location.

Procedure Find (T: in out BinaryTree; E : in Item);
-- pre - T exists.
-- post - Current Location in the tree contains E.
-- exception raised - NOT_FOUND if E cannot be found in tree
--               and Current Location remains unchanged.

Procedure PreOrderTraversal(T : in BinaryTree);
-- pre - T exists.
-- post - Current Location remains unchanged.  Action
--      specified by user supplied WhenTraversingDo
--      procedure is performed on each node in the tree in
--      preorder fashion.
```

```
Procedure InOrderTraversal(T : in BinaryTree);
-- pre - T exists.
-- post - Current Location remains unchanged.  Action
--       specified by user supplied WhenTraversingDo
--       procedure is performed on each node in the tree in
--       inorder fashion.

Procedure PostOrderTraversal(T : in BinaryTree);
-- pre - T exists.
-- post - Current Location remains unchanged.  Action
--       specified by user supplied WhenTraversingDo
--       procedure is performed on each node in the tree in
--       postorder fashion.

Procedure LevelByLevelTraversal(T : in BinaryTree);
-- pre - T exists.
-- post - Current Location remains unchanged.  Action
--       specified by user supplied WhenTraversingDo
--       procedure is performed on each node in the tree in
--       a level by level fashion starting at the level
--       containing the root node.

Private
  Type TreeInstance;
  Type BinaryTree is Access TreeInstance;

end Gen_BinaryTree;


with Unchecked_Deallocation, Generic_List;
Package body Gen_BinaryTree is

Type Node;                        -- Forward declaration.
Type Node_Pointer is Access Node;  -- Construction method of
                             -- tree.
Type Node is                      -- What tree is made up of.
  Record
    Element : Item;
    LeftChild : Node_Pointer;
    RightChild : Node_Pointer;
  end Record;

Type TreeInstance is          -- Provides access to tree and
```

```
  Record                    -- to a specific node.
    Root : Node_Pointer;
    Current : Node_Pointer;
  end Record;


package Pointer_List is
              new Generic_List(ITEM => Node_Pointer);
use Pointer_List;
                  -- Linked list that will contain
                  -- Node_Pointers.


Procedure Return_Node is
         new Unchecked_Deallocation (Node, Node_Pointer);
                 -- Used for reclaiming Node_Pointer
                 -- memory space.

Procedure Return_Tree is
    new Unchecked_Deallocation (TreeInstance, BinaryTree);
                  -- Used for reclaiming BinaryTree
                  -- memory space.

Procedure Clear (T : in out BinaryTree) is
-- pre - T exists.
-- post - T-pre exists as an empty tree.

begin
  If not Empty(T) then  --If already clear then don't do it.
    T.Current := T.Root;
    DeleteSub(T);
    Create(T);
  end If;
end Clear;

Procedure Destroy (T : in out BinaryTree) is
-- pre - T exists.
-- post - T-pre no longer exists.

begin
  Clear(T);
  Return_Tree(T);
end;
```

```
Function Full(T : in BinaryTree) Return Boolean is
-- pre - T exists.
-- post - True if the tree T can not have more items added,
--       otherwise False.

Temp : Node_Pointer;

begin
  Temp := T.Root;        -- Checks that tree exists.
  Temp := new Node;        -- Check if space is available.
  Return_Node(Temp);   -- Return space allocated by check.
  Return (false);

  Exception
    when STORAGE_ERROR =>
      Return (true);  --Full as no more memory is free.
    when others =>
      Raise;
end Full;


Function Empty(T : in BinaryTree) Return Boolean is
-- pre - T exists.
-- post - True if tree T has no items in it, otherwise False.

begin
  Return (T.Root = null);  --Empty when root points to null.
end Empty;


Procedure Create(T : in out BinaryTree) is
-- pre - None.
-- post - T exists and is empty.

TempPtr : BinaryTree;

begin
  TempPtr := new TreeInstance'(null,null);
  T := TempPtr;
end Create;
```

```
Procedure Insert (T : in out BinaryTree; E : in Item;
          C : in Character; Success : in out Boolean) is
-- pre - T exists. The size of T has not reached its maximum.
-- comment - Character 'L' or 'l' should be parameter for
--                                  left child.
--         Character 'R' or 'r' should be parameter for
--                                  right child.
-- post - If pre-T was empty then E is the content of the
--       Root node of T and Success is true.
--       Otherwise pre-Current location has a new C child
--       that contains E if Success is true and Current
--       location is the new child.
--       Success is false if C child existed prior to
--       insertion try.  Current location remains same as
--       pre if Success is false.

TempPtr : Node_Pointer;       -- Used to create a new node.

begin
  Success := True;                -- Initialization.

  If Empty(T) then        -- Insert Item into the root node.
    TempPtr := new Node'(E,null,null);
    T.Root := TempPtr;
    T.Current := T.Root;

  Else

    Case C is
      when 'L' | 'l' =>       -- Insert as new leftchild.

        If T.Current.leftchild /= null then
          Success := False; --Leftchild already existed.
        end If;

        If Success then
          TempPtr := new Node'(E,null,null);
          T.Current.leftchild := TempPtr;
          T.Current := T.Current.leftchild;
        end If;

      when 'R' | 'r' =>   -- Insert as new rightchild.
```

```
        If T.Current.rightchild /= null then
          Success := False; --Rightchild already exists.
        end If;

        If Success then
          TempPtr := new Node'(E,null,null);
          T.Current.rightchild := TempPtr;
          T.Current := T.Current.rightchild;
        end If;

      when others =>
        Success := False;
    end Case;
  end If;
end Insert;


Procedure TreeDispose (N : in out Node_Pointer) is
-- Hidden procedure that deletes sub tree rooted at N and
-- reclaims the memory space.

begin
  If N /= null then
    TreeDispose(N.leftchild); -- Delete sub tree rooted at
                              -- leftchild
    TreeDispose(N.rightchild); -- Delete sub tree rooted at
                               -- rightchild
    Return_Node(N);           -- Reclaim memory space.
  end If;
end TreeDispose;


Function FindParent(T : in BinaryTree; N : in Node_Pointer)
                    Return Node_Pointer is
-- Hidden Function that returns a pointer to the parent of
-- the node pointed to by N.  Search for parent is done level
-- by level from root.
-- SHOULD NEVER BE CALLED WITH AN EMPTY TREE.

L : List;                   -- Will hold Node_pointers.
Position : Integer;         -- Used to index into List L.
I : Node_Pointer;           -- Content of List L nodes.
TempPtr : Node_Pointer;     -- Used to traverse tree.
```

```
begin
  TempPtr := T.Root;          -- Initialization.
  Insert(L,1,TempPtr);        -- Enqueue root node in List.

  Loop               -- Level by level search for parent.
    Delete(L,1,I);        -- Serve first node of List.

    If (I.leftchild = N) or (I.rightchild = N) then
      Exit;                   -- Parent has been found.
    end If;

    If I.leftchild /= null then  --Enqueue nodes of next
      Length(L,Position);              -- level ...
      Position := Position + 1;
      Insert(L,Position,I.leftchild);
    end If;

    If I.rightChild /= null then
      Length(L,Position);
      Position := Position + 1;
      Insert(L,Position,I.rightchild);
    end If;
  end Loop;                     -- ... from left to right.

  Clear (L);             -- Reclaim memory used by List L.

  Return (I);                    -- I is parent.
end FindParent;


Procedure DeleteSub (T : in out BinaryTree) is
-- pre - T is not empty.
-- post - SubTree rooted at pre-Current location is deleted.
--       If pre-Current is root then tree no longer exists.

TempPtr : Node_Pointer; --Used to point to parent of current.

begin
  If T.Current = T.Root then
                     -- Delete left side of tree.
    If T.Root.leftchild /= null then
      T.Current := T.Root.leftchild;
      DeleteSub(T);
    end If;
                          -- Delete right side of tree.
    If T.Root.rightchild /= null then
      T.Current := T.Root.rightchild;
      DeleteSub(T);
    end If;

    T.Root := null;              -- Empty the root node.
    T.Current := T.Root;
    Return_Tree(T);

  Else

    TempPtr := FindParent(T,T.Current);

    If TempPtr.leftchild = T.Current then
      TempPtr.leftchild := null;  -- Parent's pointer to
    Else
      TempPtr.rightchild := null; --... child set to null.
    end If;

    TreeDispose(T.Current); --Reclaim memory used by
                                -- subtree.
    T.Current := TempPtr;   -- Current moves up to parent.
  end If;
end DeleteSub;


Procedure Update (T : in out BinaryTree; E : in Item) is
-- pre - T is not empty.
-- post - E is the new content of pre-Current location.

begin
  T.Current.Element := E; --Current node's content is now E.
end Update;


Procedure Retrieve (T : in BinaryTree; E : in out Item) is
-- pre - T is not empty.
-- post - E is the content of pre-Current location.

begin
  E := T.Current.Element; --E is content of the current node
```

```
end Retrieve;


Procedure Find (T: in out BinaryTree; E : in Item) is
-- pre - T exists.
-- post - Current Location in the tree contains E.
-- exception raised - NOT_FOUND if E cannot be found in tree
--                   and Current Location remains unchanged.

Found : Boolean;        -- True when item search for is found.
L : List;               -- Will hold Node_Pointers
Position : Integer;     -- Used to index into list L.
I : Node_Pointer;       -- Content of nodes in list L.

begin
  Found := false;                -- Initialization.
  If T.Root /= null then
    Insert(L,1,T.Root);

    Loop                  -- Level by level search for E.
      If Empty(L) then
        Found := False;    -- E was not found
        Exit;              -- All nodes of tree searched.
      end If;

      Delete(L,1,I);        -- Serve first node of list L.
      If I.Element = E then
        Found := true;           -- E is found.
        Exit;                    -- Terminate search.
      end If;

      If I.leftchild /= null then   -- Enqueue next
        Length(L,Position);         -- level's nodes ...
        Position := Position + 1;
        Insert(L,Position,I.leftchild);
      end If;

      If I.rightChild /= null then
        Length(L,Position);
        Position := Position + 1;
        Insert (L,Position,I.rightchild);
      end If;                    -- ... left to right.
    end Loop;
    Clear (L);          -- Reclaim memory used by list L.
  end If;

  If not Found then
    raise (NOT_FOUND);      -- Tree does not contain E.
  Else
    T.Current := I;          -- I contains E.
  end If;
end Find;


Procedure PreOrderTraversal(T : in BinaryTree) is
-- pre - T is not Empty.
-- post - Current Location remains unchanged.  Action
--       specified by user supplied WhenTraversingDo
--       procedure is performed on each node in the tree in
--       preorder fashion.

Procedure PreOrder(Ptr : in Node_Pointer) is
-- Hidden procedure that actually performs the traversal
-- recursively.

begin
  If Ptr /= null then
    WhenTraversingDo(Ptr.Element); -- Perform action  on
    PreOrder(Ptr.leftchild);              -- each node.
    PreOrder(Ptr.rightchild);
  end If;
end PreOrder;

begin
  PreOrder(T.Root);
end PreOrderTraversal;


Procedure InOrderTraversal(T : in BinaryTree) is
-- pre - T is not Empty.
-- post - Current Location remains unchanged.  Action
--       specified by user supplied WhenTraversingDo
--       procedure is performed on each node in the tree in
--       inorder fashion.
```

```
Procedure InOrder(Ptr : in Node_Pointer) is
-- Hidden procedure that actually performs traversal
-- recursively.

begin
  If Ptr /= null then
    InOrder(Ptr.leftchild);
    WhenTraversingDo(Ptr.Element); -- Perform action on
    InOrder(Ptr.rightchild);           -- each node.
  end If;
end InOrder;

begin
  InOrder(T.Root);
end InOrderTraversal;


Procedure PostOrderTraversal(T : in BinaryTree) is
-- pre - T is not Empty.
-- post - Current Location remains unchanged.  Action
--       specified by user supplied WhenTraversingDo
--       procedure is performed on each node in the tree in
--       postorder fashion.

Procedure PostOrder(Ptr : in Node_Pointer) is
-- Hidden procedure that actually performs the traversal
-- recursively.

begin
  If Ptr /= null then
    PostOrder(Ptr.leftchild);
    PostOrder(Ptr.rightchild);
    WhenTraversingDo(Ptr.Element); -- Perform action on
  end If;                          -- each node.
end PostOrder;

begin
  PostOrder(T.Root);
end PostOrderTraversal;


Procedure LevelByLevelTraversal(T : in BinaryTree) is
-- pre - T is not Empty.
-- post - Current Location remains unchanged.  Action
-- specified by user supplied WhenTraversingDo procedure is
-- performed on each node in the tree in a level by level
-- fashion starting at the level containing the root node.

L : List;
Position : Integer;
Ptr, I : Node_Pointer;

begin
  If not Empty(T) then
    Ptr := T.Root;
    Insert(L,1,Ptr);    -- Enqueue root node of tree in L.
    Loop
      If Empty(L) then
        Exit;                  -- Traversal finished.
      end If;

      Delete(L,1,I);       -- Serve first node of list L.

      WhenTraversingDo(I.Element); --Perform action on
                                        -- each node.

      If I.leftchild /= null then    -- Enqueue next
        Length(L,Position);            -- level's nodes
        Position := Position + 1;
        Insert(L,Position,I.leftchild);
      end If;

      If I.rightChild /= null then
        Length(L,Position);
        Position := Position + 1;
        Insert(L,Position,I.rightchild);
      end If;                       -- ... left to right.

    end Loop;

    Clear (L);          -- Reclaim memory used by list L.
  end If;
end LevelByLevelTraversal;
end Gen_BinaryTree;
```

Example Procedure to be Instantiated for WhenTraversingDo

```
with ada_io;
use ada_io;

procedure NodeOutput(I : Integer) is
begin
  new_line;
  put (I);
end NodeOutput;
```

Instantiation of Gen_BinaryTree

```
with ada_io;
use ada_io;
with Gen_BinaryTree, NodeOutput;

Procedure treetest is
Package Int_Tree is new Gen_BinaryTree(Item => Integer,
                  WhenTraversingDo => NodeOutput);

T : Int_Tree.BinaryTree;
Int1 : INTEGER;
Int2 : INTEGER;
Bool1 : BOOLEAN;
ch : Character;

begin
  Loop
    new_line;
    put ("0. Create.");
    new_line;
    put ("1. Full?");
    new_line;
    put ("2. Empty?");
    new_line;
    put ("3. Clear.");
    new_line;
    put ("4. Insert.");
    new_line;
```

```
put ("5. DeleteSub.");
new_line;
put ("6. Update.");
new_line;
put ("7. Retrieve.");
new_line;
put ("8. Find.");
new_line;
put ("9. Traverse.");
new_line;
put ("10. Destroy.");
new_line;
new_line;
put ("Enter number of action you desire.");
get (Int1);

Case Int1 is
  when 0 =>
    Int_Tree.Create (T);
    new_line;
    put("Tree created.");

  when 1 =>
    new_line;
    put("Checking if tree is full.");
    If Int_Tree.Full(T) then
      put(" Yes it is full.");
    Else
      put(" No it is not full.");
    end If;

  when 2 =>
    new_line;
    put ("Checking if tree is empty.");
    If Int_Tree.Empty(T) then
      put(" Yes it is empty.");
    Else
      put(" No it is not empty.");
    end If;

  when 3 =>
    new_line;
    put("Clearing the tree.");
  Int_Tree.Clear(T);
  put(" Finished clearing the tree.");

when 4 =>
  new_line;
  put("Enter integer you want to put in tree. ");
  get(Int2);
  new_line;
  Loop
    put("Enter 'l' or 'r' for left or right");
    put(" child. ");
    get(ch);
    new_line;
    Case ch is
      when 'l'|'L' =>
        Exit;
      when 'r'|'R' =>
        Exit;
      when others =>
        put ("Try again must be 'l' or 'r'.");
        new_line;
    end Case;
  end Loop;

  Int_Tree.Insert(T,Int2,ch,Bool1);
  new_line;
  If Bool1 then
    put("Insertion was successful.");
  Else
    put("Insertion was not successful.");
  end If;

when 5 =>
  new_line;
  put("Deleting sub tree rooted at current node.");
  Int_Tree.DeleteSub(T);
  put(" Finished with deletion.");

when 6 =>
  new_line;
  put("Enter integer you want current node to");
  put(" now hold. ");
  get(Int2);
  new_line;
  put("Updating current node.");
  Int_Tree.Update(T,Int2);
  put(" Finished with update.");

when 7 =>
  new_line;
  put("Retrieving content of current node.");
  Int_Tree.Retrieve(T,Int2);
  put(" Content was ");
  put (Int2);
  put (".");

when 8 =>
  new_line;
  put("Enter integer you want to find in the");
  put(" tree. ");
  get(Int2);
  new_line;
  put("Finding ");
  put(Int2);
  put(".");
  Int_Tree.Find(T,Int2);
  put(" Finished with find.");

when 9 =>
  new_line;
  Loop
    put("Enter either 'r', 'n', 'o' or 'e' for ");
    new_line;
    put ("r = pre-order, n = in-order, o = ");
    put ("post-order or e =");
    put (" level by level traversal. ");
    get(ch);
    Case ch is
      when 'r'|'R' =>
        Int_Tree.PreOrderTraversal(T);
        Exit;
      when 'n'|'N' =>
        Int_Tree.InOrderTraversal(T);
        Exit;
      when 'o'|'O' =>
        Int_Tree.PostOrderTraversal(T);
```

```
              Exit;
            when 'e'|'E' =>
              Int_Tree.LevelByLevelTraversal(T);
              Exit;
            when others =>
              put ("Try again enter either 'r',");
              put ("'n','o', or 'e'.");
              new_line;
          end Case;
        end Loop;
        new_line;
        put("Traversal finished.");

      when 10 =>
        new_line;
        put ("Destroying Tree.");
        Int_Tree.Destroy(T);
        new_line;
        put ("Tree destroyed.");

      when others =>
        EXIT;
    end Case;
  end Loop;

  Exception
    when Int_Tree.NOT_FOUND =>
      put (" What you were looking for in the tree");
      put (" isn't there.");
    when others =>
      Raise;
end Treetest;
```

APPENDIX F

BINARY SEARCH TREE

The major portion of this binary search tree algorithm is taken from [Ref. 3:pp. 239-242].

```
Generic
  Type Item is Private;

with Procedure WhenTraversingDo (Content : in Item) is <>;
-- A user defined procedure that is compatible with this
-- should be compiled for use in the instantiation of this
-- generic package.

with Function Item1LessThanItem2
    (Item1 : in Item; Item2 : in Item) Return Boolean is <>;
-- A user defined function that returns True
-- if Item1 < Item2.

Package Gen_BSTree is
  Type BSTree is Limited Private;

NOT_FOUND : Exception;

Procedure Clear (T : in out BSTree);
-- pre - T exists.
-- post - T-pre exists as an empty tree.

Procedure Destroy (T : in out BSTree);
-- pre - T exists.
-- post - T-pre exists as an empty tree.

Function Full(T : in BSTree) Return Boolean;
-- pre - T exists.
-- post - True if the tree T can not have more items added,
--       otherwise False.

Function Empty(T : in BSTree) Return Boolean;
-- pre - T exists.
-- post - True if tree T has no items in it, otherwise False.

Procedure Create(T : in out BSTree);
-- pre - None.
-- post - T exists and is empty.

Procedure Insert (T : in out BSTree; E : in Item;
                  Success : in out Boolean);
-- pre - T exists. The size of T has not reached its maximum.
-- post - If pre-T was empty then E is the content of the
--      Root node of T and Success is true.
--      Otherwise if E is not already in the tree then E
--      has been inserted in its proper position in the
--      tree.  Success is true and Current location is the
--      new child that contains E.
--      If E was in T-pre then Current location remains
--      unchanged and Success is false.

Procedure DeleteSub (T : in out BSTree);
-- pre - T is not empty.
-- post - SubTree rooted at pre-Current location is deleted.
--      If pre-Current is root then tree no longer exists.

Procedure Retrieve (T : in BSTree; E : in out Item);
-- pre - T is not empty.
-- post - E is the content of pre-Current location.

Procedure Find (T: in out BSTree; E : in Item);
-- pre - T exists.
-- post - Current Location in the tree contains E.
-- exception raised - NOT_FOUND if E cannot be found in tree
--              and Current Location points to node
--              where E would be added.

Procedure PreOrderTraversal(T : in BSTree);
-- pre - T exists.
-- post - Current Location remains unchanged.  Action
--      specified by user supplied WhenTraversingDo
--      procedure is performed on each node in the tree in
--      preorder fashion.

Procedure InOrderTraversal(T : in BSTree);
-- pre - T exists.
-- post - Current Location remains unchanged.  Action
--       specified by user supplied WhenTraversingDo
--       procedure is performed on each node in the tree in
--       inorder fashion.

Procedure PostOrderTraversal(T : in BSTree);
-- pre - T exists.
-- post - Current Location remains unchanged.  Action
--       specified by user supplied WhenTraversingDo
--       procedure is performed on each node in the tree in
--       postorder fashion.

Procedure LevelByLevelTraversal(T : in BSTree);
-- pre - T exists.
-- post - Current Location remains unchanged.  Action
--       specified by user supplied WhenTraversingDo
--       procedure is performed on each node in the tree in
--       a level by level fashion starting at the level
--       containing the root node.

Private
  Type TreeInstance;
  Type BSTree is Access TreeInstance;

end Gen_BSTree;


with Unchecked_Deallocation, Generic_List;
Package body Gen_BSTree is

Type Node;                     -- Forward declaration.
Type Node_Pointer is Access Node;  -- Construction method
                                   -- of tree.
Type Node is                   -- What tree is made up of.
  Record
    Element : Item;
    LeftChild : Node_Pointer;
    RightChild : Node_Pointer;
  end Record;

Type TreeInstance is       -- Provides access to tree and
  Record                   -- to a specific node.
    Root : Node_Pointer;
    Current : Node_Pointer;
  end Record;


package Pointer_List is
           new Generic_List(ITEM => Node_Pointer);
use Pointer_List;        -- Linked list that will contain
                                   -- Node_Pointers.


Procedure Return_Node is
         new Unchecked_Deallocation (Node,Node_Pointer);
                  -- Used for reclaiming Node_Pointer
                                   -- memory space.

Procedure Return_Tree is
        new Unchecked_Deallocation (TreeInstance,BSTree);
                      -- Used for reclaiming BSTree
                                   -- memory space.


Procedure Clear (T : in out BSTree) is
-- pre - T exists.
-- post - T-pre exists as an empty tree.

begin
  If not Empty(T) then  --If already clear then don't do it.
    T.Current := T.Root;
    DeleteSub(T);
    Create(T);
  end If;
end Clear;


Procedure Destroy (T : in out BSTree) is
-- pre - T exists.
-- post - T-pre exists as an empty tree.

begin
  Clear(T);
  Return_Tree(T);
end Destroy;
```

```
Function Full(T : in BSTree) Return Boolean is
-- pre - T exists.
-- post - True if the tree T can not have more items added,
--       otherwise False.

Temp : Node_Pointer;

begin
  Temp := T.Root;  -- Checks that tree exists.
  Temp := new Node;     -- Check if space is available.
  Return_Node(Temp);    -- Return space allocated by check.
  Return (false);

  Exception
  when STORAGE_ERROR =>
    Return (true);   --Full as no more memory is available.
  when others =>
    Raise;
end Full;

Function Empty(T : in BSTree) Return Boolean is
-- pre - T exists.
-- post - True if tree T has no items in it, otherwise False.

begin
  Return (T.Root = null);  --Empty when root points to null.
end Empty;


Procedure Create(T : in out BSTree) is
-- pre - None.
-- post - T exists and is empty.

TempPtr : BSTree;

begin
  TempPtr := new TreeInstance'(null,null);
  T := TempPtr;
end Create;


Procedure Insert (T : in out BSTree; E : in Item;
                     Success : in out Boolean) is
```

```
-- pre - T exists. The size of T has not reached its maximum.
-- post - If pre-T was empty then E is the content of the
--       Root node of T and Success is true.
--       Otherwise if E is not already in the tree then E
--       has been inserted in its proper position in the
--       tree.  Success is true and Current location is the
--       new child that contains E.  If E was in T-pre
--       then Current location remains unchanged and
--       Success is false.

TempPtr : Node_Pointer;     -- Used to create a new node.

begin
  Success := True;                -- Initialization.

  If Empty(T) then       -- Insert Item into the root node.
    TempPtr := new Node'(E,null,null);
    T.Root := TempPtr;
    T.Current := T.Root;
  Else
    Find (T, E);
    Success := False;
  end If;

  Exception
    when NOT_FOUND =>  -- Action taken to insert new node.
      If Item1LessThanItem2(E,T.Current.Element) then
        T.Current.LeftChild := new Node'(E,null,null);
        T.Current := T.Current.LeftChild;
      Else
        T.Current.RightChild := new Node'(E,null,null);
        T.Current := T.Current.RightChild;
      end If;
    when Others =>
      Raise;
end Insert;


Procedure TreeDispose (N : in out Node_Pointer) is
-- Hidden procedure that deletes sub tree rooted at N and
-- reclaims the memory space.

begin
```

```
  If N /= null then
    TreeDispose(N.leftchild);  --Delete sub tree rooted at
                               -- leftchild.
    TreeDispose(N.rightchild); --Delete sub tree rooted at
                               -- rightchild.
    Return_Node(N);     -- Reclaim memory space.
  end If;
end TreeDispose;


Function FindParent(T : in BSTree; N : in Node_Pointer)
                           Return Node_Pointer is
-- Hidden Function that returns a pointer to the parent of
-- the node pointed to by N.  Search for parent is done level
-- by level from root.
-- SHOULD NEVER BE CALLED WITH AN EMPTY TREE.

L : List;                 -- Will hold Node_pointers.
Position : Integer;           -- Used to index into List L.
I : Node_Pointer;             -- Content of List L nodes.
TempPtr : Node_Pointer;         -- Used to traverse tree.

begin
  TempPtr := T.Root;          -- Initialization.
  Insert(L,1,TempPtr);       -- Enqueue root node in List.

  Loop                -- Level by level search for parent.
    Delete(L,1,I);    -- Serve first node of List.

    If (I.leftchild = N) or (I.rightchild = N) then
      Exit;                  -- Parent has been found.
    end If;

    If I.leftchild /= null then    -- Enqueue nodes of
      Length(L,Position);              -- next level ...
      Position := Position + 1;
      Insert(L,Position,I.leftchild);
    end If;

    If I.rightChild /= null then
      Length(L,Position);
      Position := Position + 1;
      Insert(L,Position,I.rightchild);
    end If;                 -- ... from left to right.
  end Loop;

  Clear (L);            -- Reclaim memory used by List L.

  Return (I);                -- I is parent.
end FindParent;


Procedure DeleteSub (T : in out BSTree) is
-- pre - T is not empty.
-- post - SubTree rooted at pre-Current location is deleted.
--       If pre-Current is root then tree no longer exists.

TempPtr : Node_Pointer; --Used to point to parent of current.

begin
  If T.Current = T.Root then
                      -- Delete left side of tree.
    If T.Root.leftchild /= null then
      T.Current := T.Root.leftchild;
      DeleteSub(T);
    end If;
                      -- Delete right side of tree.
    If T.Root.rightchild /= null then
      T.Current := T.Root.rightchild;
      DeleteSub(T);
    end If;

    T.Root := null;            -- Empty the root node.
    T.Current := T.Root;
    Return_Tree(T);

  Else

    TempPtr := FindParent(T,T.Current);

    If TempPtr.leftchild = T.Current then
      TempPtr.leftchild := null;  -- Parent's pointer
    Else                          -- to child
      TempPtr.rightchild := null;      -- set to null.
    end If;
    TreeDispose(T.Current);  -- Reclaim memory used by
                             -- subtree.
    T.Current := TempPtr;  --Current moves up to parent of
                           -- deleted subtree.
  end If;
end DeleteSub;


Procedure Retrieve (T : in BSTree; E : in out Item) is
-- pre - T is not empty.
-- post - E is the content of pre-Current location.

begin
  E := T.Current.Element;  --E is the content of the
end Retrieve;               -- current node.


Procedure Find (T: in out BSTree; E : in Item) is
-- pre - T exists.
-- post - Current Location in the tree contains E.
-- exception raised - NOT_FOUND if E cannot be found in tree
--                    and Current Location points to node
--                    where E would be added.

Found : Boolean;   -- True when item searched for is found.
SearchPtr : Node_Pointer;
Parent : Node_Pointer;

begin
  SearchPtr := T.Root;          -- Initialization.
  Found := false;               -- Initialization.

  While SearchPtr /= null
  Loop
    Parent := SearchPtr;

    If SearchPtr.Element = E then
      T.Current := SearchPtr;     -- Search completed.
      Found := True;
      Exit;

    Else
```

```
      If Item1LessThanItem2(E, SearchPtr.Element) then
        SearchPtr := SearchPtr.LeftChild;
      Else
        SearchPtr := SearchPtr.RightChild;
      end If;
    end If;
  end Loop;

  If not Found then
    T.Current := Parent;
    raise (NOT_FOUND);      -- Tree does not contain E.
  end If;

end Find;


Procedure PreOrderTraversal(T : in BSTree) is
-- pre - T is not Empty.
-- post - Current Location remains unchanged.  Action
--       specified by user supplied WhenTraversingDo
--       procedure is performed on each node in the tree in
--       preorder fashion.

Procedure PreOrder(Ptr : in Node_Pointer) is
-- Hidden procedure that actually performs the traversal
-- recursively.

begin
  If Ptr /= null then
    WhenTraversingDo(Ptr.Element); -- Perform action on
    PreOrder(Ptr.leftchild);           -- each node.
    PreOrder(Ptr.rightchild);
  end If;
end PreOrder;

begin
  PreOrder(T.Root);
end PreOrderTraversal;


Procedure InOrderTraversal(T : in BSTree) is
-- pre - T is not Empty.
-- post - Current Location remains unchanged.  Action
--      specified by user supplied WhenTraversingDo
--      procedure is performed on each node in the tree in
--      inorder fashion.


Procedure InOrder(Ptr : in Node_Pointer) is
-- Hidden procedure that actually performs traversal
-- recursively.

begin
  If Ptr /= null then
    InOrder(Ptr.leftchild);
    WhenTraversingDo(Ptr.Element); -- Perform action on
    InOrder(Ptr.rightchild);             -- each node.
  end If;
end InOrder;

begin
  InOrder(T.Root);
end InOrderTraversal;


Procedure PostOrderTraversal(T : in BSTree) is
-- pre - T is not Empty.
-- post - Current Location remains unchanged.  Action
--      specified by user supplied WhenTraversingDo
--      procedure is performed on each node in the tree in
--      postorder fashion.

Procedure PostOrder(Ptr : in Node_Pointer) is
-- Hidden procedure that actually performs the traversal
-- recursively.

begin
  If Ptr /= null then
    PostOrder(Ptr.leftchild);
    PostOrder(Ptr.rightchild);
    WhenTraversingDo(Ptr.Element); -- Perform action on
  end If;                            -- each node.
end PostOrder;

begin
  PostOrder(T.Root);
```

```
end PostOrderTraversal;


Procedure LevelByLevelTraversal(T : in BSTree) is
-- pre - T is not Empty.
-- post - Current Location remains unchanged.  Action
--      specified by user supplied WhenTraversingDo
--      procedure is performed on each node in the tree in
--      a level by level fashion starting at the level
--      containing the root node.

L : List;
Position : Integer;
Ptr, I : Node_Pointer;

begin
  If not Empty(T) then
    Ptr := T.Root;
    Insert(L,1,Ptr);   -- Enqueue root node of tree in L.
    Loop
      If Empty(L) then
        Exit;                  -- Traversal finished.
      end If;

      Delete(L,1,I);      -- Serve first node of list L.

      WhenTraversingDo(I.Element);  -- Perform action on
                                    -- each node.

      If I.leftchild /= null then  --Enqueue next
        Length(L,Position);        -- level's nodes ...
        Position := Position + 1;
        Insert(L,Position,I.leftchild);
      end If;

      If I.rightChild /= null then
        Length(L,Position);
        Position := Position + 1;
        Insert(L,Position,I.rightchild);
      end If;                  -- ... left to right.

    end Loop;
    Clear (L);          -- Reclaim memory used by list L.
  end If;
end LevelByLevelTraversal;
end Gen_BSTree;
```

Example Function to be Instantiated for Item1LessThanItem2

```
Function IntegerLessThan(Item1 : in Integer;
                Item2 : in Integer) Return Boolean is

begin
  Return (Item1 < Item2);
end IntegerLessThan;
```

Instantiation of Gen_BSTree

```
with ada_io;
use ada_io;
with Gen_BSTree, NodeOutput, IntegerLessThan;
Procedure bsttest is
Package Int_BST is new Gen_BSTree(Item => Integer,
              WhenTraversingDo => NodeOutput,
              Item1LessThanItem2 => IntegerLessThan);
   -- NodeOutput details can be found in Appendix E.

T : Int_BST.BSTree;
Int1 : INTEGER;
Int2 : INTEGER;
Bool1 : BOOLEAN;
ch : Character;

begin
  Loop
    new_line;
    put ("0. Create.");
    new_line;
```

```
put ("1. Full?");
new_line;
put ("2. Empty?");
new_line;
put ("3. Clear.");
new_line;
put ("4. Insert.");
new_line;
put ("5. DeleteSub.");
new_line;
put ("6. Retrieve.");
new_line;
put ("7. Find.");
new_line;
put ("8. Traverse.");
new_line;
put ("9. Destroy.");
new_line;
new_line;
put ("Enter number of action you desire.");
get (Int1);

Case Int1 is
  when 0 =>
    Int_BST.Create (T);
    new_line;
    put("Tree created.");

  when 1 =>
    new_line;
    put("Checking if tree is full.");
    If Int_BST.Full(T) then
      put(" Yes it is full.");
    Else
      put(" No it is not full.");
    end If;

  when 2 =>
    new_line;
    put ("Checking if tree is empty.");
    If Int_BST.Empty(T) then
      put(" Yes it is empty.");
    Else
```

```
      put(" No it is not empty.");
    end If;

  when 3 =>
    new_line;
    put("Clearing the tree.");
    Int_BST.Clear(T);
    put(" Finished clearing the tree.");

  when 4 =>
    new_line;
    put("Enter integer you want to put in tree. ");
    get(Int2);
    new_line;
    Int_BST.Insert(T,Int2,Bool1);
    new_line;
    If Bool1 then
      put("Insertion was successful.");
    Else
      put("Insertion was not successful.");
    end If;

  when 5 =>
    new_line;
    put("Deleting sub tree rooted at current node.");
    Int_BST.DeleteSub(T);
    put(" Finished with deletion.");

  when 6 =>
    new_line;
    put("Retrieving content of current node.");
    Int_BST.Retrieve(T,Int2);
    put(" Content was ");
    put (Int2);
    put (".");

  when 7 =>
    new_line;
    put("Enter integer you want to find in the");
    put(" tree. ");
    get(Int2);
    new_line;
    put("Finding ");
  put(Int2);
  put(".");
  Int_BST.Find(T,Int2);
  put(" Finished with find.");

when 8 =>
  new_line;
  Loop
    put("Enter either 'r', 'n', 'o' or 'e' for ");
    new_line;
    put ("r = pre-order, n = in-order, o = ");
    put ("post-order or e =");
    put (" level by level traversal. ");
    get(ch);
    Case ch is
      when 'r'|'R' =>
        Int_BST.PreOrderTraversal(T);
        Exit;
      when 'n'|'N' =>
        Int_BST.InOrderTraversal(T);
        Exit;
      when 'o'|'O' =>
        Int_BST.PostOrderTraversal(T);
        Exit;
      when 'e'|'E' =>
        Int_BST.LevelByLevelTraversal(T);
        Exit;
      when others =>
        put ("Try again enter either 'r','n'");
        put (",'o', or 'e'.");
        new_line;
    end Case;
  end Loop;
  new_line;
  put("Traversal finished.");

when 9=>
  new_line;
  put("Destroying Tree.");
  Int_BST.Destroy(T);
  new_line;
  put ("Tree destroyed.");
```

```
        when others =>
          EXIT;
      end Case;
    end Loop;

    Exception
      when Int_BST.NOT_FOUND =>
        put (" What you were looking for in the tree");
        put (" isn't there.");
      when others =>
        Raise;
end bsttest;
```

APPENDIX G

AVL TREE

The following algorithm is a combination of the AVL tree algorithm found in [Ref. 3:pp. 259-260] and in [Ref. 11:pp. 225-227].

```
Generic
  Type Item is Private;

with Procedure WhenTraversingDo (Content : in Item) is <>;
-- A user defined procedure that is compatible with this
-- should be compiled for use in the instantiation of this
-- generic package.

with Function Item1LessThanItem2
   (Item1 : in Item; Item2 : in Item) Return Boolean is <>;
-- A user defined function that returns True if Item1 < Item2.

Package Gen_AVLTree is
  Type AVLTree is Limited Private;

NOT_FOUND : Exception;
DELETE_NOT_FOUND : Exception;

Procedure Clear (T : in out AVLTree);
-- pre - T exists.
-- post - T-pre exists as an empty tree.

Procedure Destroy (T : in out AVLTree);
-- pre - T exists.
-- post - T-pre no longer exists.

Function Full(T : in AVLTree) Return Boolean;
-- pre - T exists.
-- post - True if the tree T can not have more items added,
-- otherwise False.
```

```
Function Empty(T : in AVLTree) Return Boolean;
-- pre - T exists.
-- post - True if tree T has no items in it, otherwise False.

Procedure Create(T : in out AVLTree);
-- pre - None.
-- post - T exists and is empty.

Procedure Insert (T : in out AVLTree; E : in Item;
                    Success : in out Boolean);
-- pre - T exists. The size of T has not reached its maximum.
-- post - If pre-T was empty then E is the content of the
--      Root node of T and Success is true.
--      Otherwise if E is not already in the tree then E
--      has been inserted in its proper position in the
--      tree.  Success is true and Current location is the
--      new node that contains E. If E was in T-pre then
--      Current location remains unchanged and Success is
--      false.  In all cases T remains an AVL tree.

Procedure Delete (T : in out AVLTree; E : in Item);
-- pre - T is not empty.
-- post - Item E is deleted from T and T remains an AVL tree.

Procedure Retrieve (T : in AVLTree; E : in out Item);
-- pre - T is not empty.
-- post - E is the content of pre-Current location.

Procedure Find (T: in out AVLTree; E : in Item);
-- pre - T exists.
-- post - Current Location in the tree contains E.
-- exception raised - NOT_FOUND if E cannot be found in tree
--              and Current Location points to node
--              where E would be added.

Procedure PreOrderTraversal(T : in AVLTree);
-- pre - T exists.
-- post - Current Location remains unchanged.  Action
--      specified by user supplied WhenTraversingDo
--      procedure is performed on each node in the tree in
--      preorder fashion.
```

```
Procedure InOrderTraversal(T : in AVLTree);
-- pre - T exists.
-- post - Current Location remains unchanged.  Action
--       specified by user supplied WhenTraversingDo
--       procedure is performed on each node in the tree in
--       inorder fashion.

Procedure PostOrderTraversal(T : in AVLTree);
-- pre - T exists.
-- post - Current Location remains unchanged.  Action
--       specified by user supplied WhenTraversingDo
--       procedure is performed on each node in the tree in
--       postorder fashion.

Procedure LevelByLevelTraversal(T : in AVLTree);
-- pre - T exists.
-- post - Current Location remains unchanged.  Action
--       specified by user supplied WhenTraversingDo
--       procedure is performed on each node in the tree in
--       a level by level fashion starting at the level
--       containing the root node.

Private
  Type TreeInstance;
  Type AVLTree is Access TreeInstance;

end Gen_AVLTree;



with Unchecked_Deallocation, Generic_List;
Package body Gen_AVLTree is

Type Node;                          -- Forward declaration.
Type Node_Pointer is Access Node;    -- Construction method
                                       -- of tree.
Type High is (left, equal, right);   -- Records Balance
                                    -- factor of tree.

Type Node is                     -- What tree is made up of.
  Record
    Element : Item;
    Balance : High;
    LeftChild : Node_Pointer;
    RightChild : Node_Pointer;
  end Record;

Type TreeInstance is       -- Provides access to tree and
  Record                   -- to a specific node.
    Root : Node_Pointer;
    Current : Node_Pointer;
  end Record;


Package Pointer_List is
            new Generic_List(ITEM => Node_Pointer);
use Pointer_List;          -- Linked list that will contain
                               -- Node_Pointers.


Procedure Return_Node is
        new Unchecked_Deallocation (Node,Node_Pointer);
                  -- Used for reclaiming Node_Pointer
                               -- memory space.


Procedure Return_Tree is
        new Unchecked_Deallocation (TreeInstance,AVLTree);
                      -- Used for reclaiming AVLTree
                               -- memory space.



Procedure Clear (T : in out AVLTree) is
-- pre - T exists.
-- post - T-pre exists as an empty tree.

begin
  While not Empty(T)  -- If already clear then don't do it.
  Loop
    T.Current := T.Root;
    Delete(T, T.Root.Element);
  end Loop;
end Clear;
```

```
Procedure Destroy (T : in out AVLTree) is
-- pre - T exists.
-- post - T-pre no longer exists.

begin
  Clear(T);          -- Reclaims memory space used by tree.
  Return_Tree(T);             -- Reclaims instance of tree.
end Destroy;


Function Full(T : in AVLTree) Return Boolean is
-- pre - T exists.
-- post - True if the tree T can not have more items added,
--        otherwise False.

Temp : Node_Pointer;

begin
  Temp := T.Root;  -- Checks that tree exists.
  Temp := new Node;       -- Check if space is available.
  Return_Node(Temp);   -- Return space allocated by check.
  Return (false);

  Exception
    when STORAGE_ERROR =>
      Return (true);  --Full as no more memory is free.
    when others =>
      Raise;
end Full;


Function Empty(T : in AVLTree) Return Boolean is
-- pre - T exists.
-- post - True if tree T has no items in it, otherwise False.

begin
  Return (T.Root = null);  --Empty when root points to null.
end Empty;

Procedure Create(T : in out AVLTree) is
-- pre - None.
-- post - T exists and is empty.
```

```
TempPtr : AVLTree;

begin
  TempPtr := new TreeInstance'(null,null);
  T := TempPtr;
end Create;


Function FindParent(T : in AVLTree; N : in Node_Pointer)
                          Return Node_Pointer is
-- Hidden Function that returns a pointer to the parent of
-- the node pointed to by N.  Search for parent is a binary
-- search.
-- SHOULD NEVER BE CALLED WITH AN EMPTY TREE.
-- SHOULD NEVER BE CALLED WITH N POINTING AT ROOT OF TREE.

TempPtr : Node_Pointer;        -- Used to traverse tree.

begin
  TempPtr := T.Root;

  Loop
    While Item1LessThanItem2(TempPtr.Element, N.Element)
    Loop               -- Search Right subtree
      If TempPtr.RightChild = N then
        Return(TempPtr);          -- Parent found!
      end If;
      TempPtr := TempPtr.RightChild;
    end Loop;

    While Item1LessThanItem2(N.Element, TempPtr.Element)
    Loop               -- Search Left subtree
      If TempPtr.LeftChild = N then
        Return (TempPtr);         -- Parent found!
      end If;
      TempPtr := TempPtr.LeftChild;
    end Loop;
  end Loop;
end FindParent;


Procedure SetPointers (T : in AVLTree;
             p1, p2, p3, p4 : in out Node_Pointer) is
```

```
-- Hidden procedure that sets Node_Pointers as follows:
-- p1 --> parent of pivot.
-- p2 --> pivot.
-- p3 --> child of pivot along the search path to new node
-- p4 --> child of p3 along the search path to new node

Temp : Node_Pointer;

Begin
  Temp := T.Root;                 -- Initialization.

  While Temp.Element /= T.Current.Element
  Loop
    Case Temp.Balance is
      when left | right => --Sets p2 to closest unbalanced
        p2 := Temp;             -- ancestor of new node.

                  -- Sets p3 to child of p2 on path to
                                 -- new node.
        If Item1LessThanItem2
               (T.Current.Element, Temp.Element) then
          p3 := p2.LeftChild;
          Temp := p3;
        Else
          p3 := p2.RightChild;
          Temp := p3;
        end If;

                  -- Sets p4 to child of p3 on path to
                                 -- new node.
        If Item1LessThanItem2
               (T.Current.Element, p3.Element) then
          p4 := p3.LeftChild;
        Else
          p4 := p3.RightChild;
        end If;

      when equal =>  -- Moves Temp down tree towards new
                                  -- node.
        If Item1LessThanItem2
               (T.Current.Element, Temp.Element) then
          Temp := Temp.LeftChild;
        Else
```

```
          Temp := Temp.RightChild;
        end If;

    end Case;
  end Loop;

                  -- If pivot exists and is not the root
                       -- then find the parent of pivot.
  If (p2 /= T.Root) and (p2 /= null) then
    p1 := FindParent(T, p2);
  end If;
end SetPointers;


Procedure Reset (T : in out AVLTree;
                          p : in out Node_Pointer) is
-- Rebalances nodes on search path from p to the new node.
-- Called only when the insertion does not cause a rotation.

begin                  -- Correct balance from p down to
                                   -- the new node.
  While T.Current.Element /= p.Element
  Loop
    If Item1LessThanItem2
                (T.Current.Element, p.Element) then
      Case p.Balance is
        when equal =>
          p.Balance := left;  -- New node is left of p.

        when right =>
          p.Balance := equal;  -- New node balances p.

        when others =>       -- p.Balance cannot = left.
          null;
      end Case;
      p := p.LeftChild;     -- Proceed towards new node.

    Else

      Case p.Balance is
        when equal =>
          p.Balance := right;  --New node is right of p.
        when left =>
          p.Balance := equal;   -- New node balances p.

        when others =>      -- p.Balance cannot = right.
          null;
      end Case;
      p := p.RightChild;     -- Proceed towards new node.

    end If;
  end Loop;
  p := T.Root;   -- Prevents loss of tree if root was pivot.
end Reset;


Function Short (p2 : in Node_Pointer;
                   E : in Item) Return Boolean is
-- Returns true if the new node containing E is on the Short
-- side of p2.  Otherwise returns false.

begin
  Case p2.Balance is
    when left =>
      If Item1LessThanItem2(E, p2.Element) then
        Return (False); --New node is on tall side of p2.
      Else
        Return (True); --New node is on short side of p2.
      end If;

    when right =>
      If Item1LessThanItem2 (E, p2.Element) then
        Return (True); --New node is on short side of p2.
      Else
      end If;

    when others =>      -- p2.Balance cannot = equal.
      null;
  end Case;
end Short;


Function Single (p2, p3, p4 : Node_Pointer) Return Boolean is
-- Returns true if a single rotation is required to balance
-- the tree.  Otherwise returns false.
```

```
begin
  Case p2.Balance is
    when right =>
      If Item1LessThanItem2 (p3.Element, p4.Element) then
        Return (True);          -- p3 is right of p2 and
                            -- p4 is right of p3
      Else
        Return (False);
      end If;

    when left =>
      If Item1LessThanItem2 (p4.Element, p3.Element) then
        Return (True);          -- p3 is left of p2 and
                            -- p4 is left of p3
      Else
        Return (False);
      end If;

    when others =>          -- p2.Balance cannot = equal.
      null;

  end Case;
end Single;


Procedure SingleRot(p1, p2, p3, p4 : in Node_Pointer;
                        T : in out AVLTree) is
-- Performs a single rotation on T below node p1, using p2 as
-- the pivot.

begin
  If p1 /= null then              -- p2 is not the root.
    If p1.LeftChild = p2 then
      p1.LeftChild := p3;  --Connects p3 to same position
    Else                        -- underneath p1 that
      p1.RightChild := p3;         -- p2 was occupying.
    end If;
  end If;

  If p2.LeftChild = p3 then

    If (p4.Element /= T.Current.Element) and
                             (p4 /= null) then
      p4.Balance := Left;
    end If;

    p2.LeftChild := p3.RightChild; -- Breaks tree apart and
    p3.RightChild := p2;            -- puts it together
                                    -- again balanced.

    If (p2.LeftChild = null) and
                    (p2.RightChild /= null) then
      p2.Balance := Right;
    Else
      p2.Balance := Equal;
    end If;

  Else

    If (p4.Element /= T.Current.Element) and
                                    (p4 /= null) then
      p4.Balance := Right;
    end If;

    p2.RightChild := p3.LeftChild; -- Breaks tree apart and
    p3.LeftChild := p2;             -- puts it together
                                    -- again balanced.
    If (p2.RightChild = null) and
                    (p2.LeftChild /= null) then
      p2.Balance := Left;
    Else
      p2.Balance := Equal;
    end If;

  end If;

  If p1 = null then   -- p2 must have been root therefore
    T.Root := p3;     -- Root must be updated to be p3.
  end If;
end SingleRot;


Procedure DoubleRot(p1, p2, p3, p4 : in Node_Pointer;
                          T : in out AVLTree) is
-- Performs a double rotation on T below node p1, using p2 as
-- the pivot.
```

```
begin
 If p1 /= null then              -- p2 is not the root.
   If p1.LeftChild = p2 then
     p1.LeftChild := p4;  --Connects p4 to same position
   Else                      -- underneath p1 that
     p1.RightChild := p4;          -- p2 was occupying.
   end If;
 end If;

 If p2.LeftChild = p3 then
   p3.RightChild := p4.LeftChild; -- Breaks off subtrees
   p2.LeftChild := p4.RightChild; -- of p4 and reconnects
                               -- them to p2 and p3.

   If p2.RightChild /= null then
     If Item1LessThanItem2
               (p4.Element, T.Current.Element) then
       p2.Balance := Equal;  -- p2's new left subtree
                           -- contains new node.
     Else
       p2.Balance := Right; --p2's new left subtree does
     end If;               --not contain the new node.

   Else

     p2.Balance := Equal; -- p4 is the new node
                     -- therefore p2 has no children.
   end If;

   If p3.LeftChild /= null then
     If Item1LessThanItem2
              (T.Current.Element, p4.Element) then
       p3.Balance := Equal; -- p3's new right subtree
                           -- contains new node
     Else
       p3.Balance := Left;  -- p3's new right subtree
     end If;              -- does not contain the new
                                    -- node.

   Else

     If Item1LessThanItem2
               (T.Current.Element, p4.Element) then
```

```
      p3.Balance := Right; -- p3 has only one child and
                                    -- it is new node.
    Else
      p3.Balance := Equal; -- p3 has no children.
    end If;
  end If;

  p4.LeftChild := p3; --Reconnects subtrees rooted at p3
  p4.RightChild := p2; -- and p2 to p4 which is now
                                       -- balanced.

Else

  p3.LeftChild := p4.RightChild; -- Breaks off subtrees
  p2.RightChild := p4.LeftChild; -- of p4 and reconnects
                               -- them to p2 and p3.

  If p2.LeftChild /= null then
    If Item1LessThanItem2
              (T.Current.Element, p4.Element) then
      p2.Balance := Equal; -- p2's new right subtree
                              -- contains new node
    Else
      p2.Balance := Left;  -- p2's new right subtree
    end If;               -- does not contain the new
                                         -- node.

  Else

    p2.Balance := Equal;    -- p4 is the new node and
                -- therefore p2 has no children.
  end If;

  If p3.RightChild /= null then
    If Item1LessThanItem2
              (p4.Element, T.Current.Element) then
      p3.Balance := Equal; -- p3's new left subtree
                      -- contains new node.
    Else

      p3.Balance := Right; -- p3's new left subtree
    end If;               -- does not contain the new
                      -- node.
    Else

      If Item1LessThanItem2
                (p4.Element, T.Current.Element) then
        p3.Balance := Left; -- p3 has only one child and
                        -- it is new node
      Else
        p3.Balance := Equal;-- p3 has no children
      end If;
    end If;

    p4.LeftChild := p2;  --Reconnects subtrees rooted at p2
    p4.RightChild := p3;     -- and p3 to p4 which is now
                             -- balanced.

  end If;

  p4.Balance := Equal;

  If p1 = null then  -- p2 must have been the root therefore
    T.Root := p4;        -- Root must be updated to be p4.
  end If;
end DoubleRot;


Procedure Insert (T : in out AVLTree; E : in Item;
                        Success : in out Boolean) is

-- pre - T exists. The size of T has not reached its maximum.
-- post - If pre-T was empty then E is the content of the
--       Root node of T and Success is true.
--       Otherwise if E is not already in the tree then E
--       has been inserted in its proper position in the
--       tree.  Success is true and Current location is the
--       new child that contains E.  If E was in T-pre then
--       Current location remains unchanged and Success is
--       false.

TempPtr : Node_Pointer;        -- Used to create a new node.
p1, p2, p3, p4 : Node_Pointer := null;

begin
  Success := True;               -- Initialization.
```

```
If Empty(T) then        -- Insert Item into the root node.
  TempPtr := new Node'(E,Equal,null,null);
  T.Root := TempPtr;
  T.Current := T.Root;
Else
  Find (T, E);  -- If not found then exception NOT_FOUND
                                  -- is raised.

  Success := False;    -- E was found in T therefore
                       -- insertion failed.
end If;

Exception

  When NOT_FOUND =>   -- Action taken to insert new node.

    If Item1LessThanItem2(E,T.Current.Element) then
      T.Current.LeftChild :=
                    new Node'(E,Equal,null,null);
      T.Current := T.Current.LeftChild;

    Else

      T.Current.RightChild :=
                    new Node'(E,Equal,null,null);
      T.Current := T.Current.RightChild;
    end If;

    SetPointers (T, p1, p2, p3, p4);

    If p2 = null then          -- No pivot exists,
      Reset (T, T.Root);       -- just adjust balances.

    Else

      If Short (p2, E) then  -- New node was added to
        If p2 = T.Root then           -- short side.
          Reset (T, T.Root);
        Else
          Reset (T, p2);
        end If;

      Else       -- New node was added to tall side.
```

```
              If Single (p2, p3, p4) then
                SingleRot (p1, p2, p3, p4, T);
              Else
                DoubleRot (p1, p2, p3, p4, T);
              end If;
            end If;
          end If;

      when Others =>
            Raise;
end Insert;


Procedure Delete (T : in out AVLTree; E : in Item) is
-- pre - T is not empty.
-- post - Item E is deleted from T and T remains an AVL tree.

H : Boolean := False;


Procedure Delete1 (E : in Item; P : in out Node_Pointer;
                         H : in out Boolean) is
Q : Node_Pointer := null;
NewRoot : Node_Pointer := null;


Procedure BalanceL (P : in out Node_Pointer;
                        H : in out Boolean) is
-- This is a hidden procedure called only from Delete.
-- Rebalances after left subtree has shrunk.

P1, P2 : Node_Pointer := null;
B1, B2 : High;

begin
  Case P.Balance is
    when left =>
      P.Balance := equal;

    when equal =>
      P.Balance := right;
      H := false;
```

```
when right =>
  P1 := P.RightChild;
  B1 := P1.Balance;

  If (B1 = equal) or (B1 = right) then
                    -- Single right/right rotation
    P.RightChild := P1.LeftChild;
    P1.LeftChild := P;
    If B1 = equal then
      P.Balance := right;
      P1.Balance := left;
      H := false;

    Else

      P.Balance := equal;
      P1.Balance := equal;
    end If;

    P := P1;

  Else                  -- Double right/left rotation

    P2 := P1.LeftChild;
    B2 := P2.Balance;
    P1.LeftChild := P2.RightChild;
    P2.RightChild := P1;
    P.RightChild := P2.LeftChild;
    P2.LeftChild := P;

    If B2 = right then
      P.Balance := left;

    Else

      P.Balance := equal;
    end If;

    If B2 = left then
      P1.Balance := right;

    Else
```

```
          P1.Balance := equal;
        end If;

        P := P2;
        P2.Balance := equal;

      end If;
  end Case;

end BalanceL;


Procedure BalanceR(P : in out Node_Pointer;
                       H : in out Boolean) is
-- This is a hidden procedure called only from Delete.
-- Rebalances after right subtree has shrunk.

P1,P2 : Node_Pointer := null;
B1,B2 : High;

begin
  Case P.Balance is
    when right =>
      P.Balance := equal;

    when equal =>
      P.Balance := left;
      H := false;

    when left =>
      P1 := P.LeftChild;
      B1 := P1.Balance;

      If (B1 = left) or (B1 = equal) then
                     -- Single left/left rotation
        P.LeftChild := P1.RightChild;
        P1.RightChild := P;

        If B1 = equal then
          P.Balance := left;
          P1.Balance := right;
          H := False;
          Else

            P.Balance := equal;
            P1.Balance := equal;
          end If;

          P := P1;

        Else              -- Double left/right rotation

          P2 := P1.RightChild;
          B2 := P2.Balance;
          P1.RightChild := P2.LeftChild;
          P2.LeftChild := P1;
          P.LeftChild := P2.RightChild;
          P2.RightChild := P;

          If B2 = left then
            P.Balance := right;

          Else

            P.Balance := equal;
          end If;

          If B2 = right then
            P1.Balance := left;

          Else

            P1.Balance := equal;
          end If;

          P := P2;
          P2.Balance := equal;

        end If;

    end Case;
end BalanceR;
```

```
Procedure Del(R : in out Node_Pointer; H : in out Boolean) is
-- This is a hidden procedure called only from Delete.
-- Finds in order predecessor of node to be deleted and moves
-- it in to the deleted nodes position.

begin
  If R.RightChild /= null then --Finds inorder predecessor
    Del(R.RightChild, H);            -- of deleted node.

    If H then
      BalanceR(R,H);
    end If;

  Else

    Q.Element := R.Element; -- Moves in order predecessor
                            -- contents up to replace node
                            -- whose contents are deleted.
    Q := R;

    R := R.LeftChild; --Retain child if any of predecessor
                                              -- node.
    H := true;     -- Need-to-Balance flag set.
  end If;
end Del;

begin -- Delete1
  If P = null then
    Raise (DELETE_NOT_FOUND); --E does not exist in tree.

  Elsif Item1LessThanItem2(E,P.Element) then
    Delete1(E,P.LeftChild,H); -- Move down one level
                    -- towards node that could contain E.
    If H then
      BalanceL(P,H);
    end If;

  Elsif Item1LessThanItem2(P.Element, E) then
    Delete1(E,P.RightChild,H); --Move down one level
                      -- towards node that could contain E.
    If H then
      BalanceR(P,H);
    end If;
  Else              -- P points to node that contains E.
    Q := P;
    If Q.RightChild = null then  --At most 1 child, move it
      P := Q.LeftChild;                 -- up the tree.
      H := true;
    Elsif Q.LeftChild = null then  --At most 1 child, move
      P := Q.RightChild;             -- it up the tree.
      H := true;        -- Need-to-Balance flag set.
    Else
      Del(Q.LeftChild,H); --Start search to find in order
                  -- predecessor of node being deleted.
      If H then
        BalanceL(P,H);
      end If;
    end If;

    Return_Node(Q);  -- Reclaim memory of one deleted node.
  end If;
end Delete1;

begin -- Delete
  Delete1(E,T.Root,H);  -- Interface to Recursive algorithm
end Delete;


Procedure Retrieve (T : in AVLTree; E : in out Item) is
-- pre - T is not empty.
-- post - E is the content of pre-Current location.

begin
  E := T.Current.Element;  --E is the content of the
end Retrieve;                    -- current node.


Procedure Find (T: in out AVLTree; E : in Item) is
-- pre - T exists.
-- post - Current Location in the tree contains E.
-- exception raised - NOT_FOUND if E cannot be found in tree
--                  and Current Location points to node
--                  where E would be added.

Found : Boolean;    -- True when item searched for is found.
SearchPtr : Node_Pointer;
```

```
Parent : Node_Pointer;

begin
  SearchPtr := T.Root;              -- Initialization.
  Found := false;                   -- Initialization.

  While SearchPtr /= null
  Loop

    Parent := SearchPtr;
    If SearchPtr.Element = E then
      T.Current := SearchPtr;     -- Search completed.
      Found := True;
      Exit;

    Else

      If Item1LessThanItem2(E, SearchPtr.Element) then
        SearchPtr := SearchPtr.LeftChild;
      Else
        SearchPtr := SearchPtr.RightChild;
      end If;

    end If;

  end Loop;

  Return_Node(SearchPtr);           -- Reclaims memory.

  If not Found then
    T.Current := Parent;
    Raise (NOT_FOUND);              -- Tree does not contain E.
  end If;
end Find;


Procedure PreOrderTraversal(T : in AVLTree) is
-- pre - T exists.
-- post - Current Location remains unchanged.  Action
--       specified by user supplied WhenTraversingDo
--       procedure is performed on each node in the tree in
--       preorder fashion.
```

```
Procedure PreOrder(Ptr : in Node_Pointer) is
-- Hidden procedure that actually performs the traversal
-- recursively.

begin
  If Ptr /= null then
    WhenTraversingDo(Ptr.Element); -- Perform action on
    PreOrder(Ptr.LeftChild);          -- each node.
    PreOrder(Ptr.RightChild);
  end If;
end PreOrder;

begin
  PreOrder(T.Root);
end PreOrderTraversal;


Procedure InOrderTraversal(T : in AVLTree) is
-- pre - T exists.
-- post - Current Location remains unchanged.  Action
--      specified by user supplied WhenTraversingDo
--      procedure is performed on each node in the tree in
--      inorder fashion.

Procedure InOrder(Ptr : in Node_Pointer) is
-- Hidden procedure that actually performs traversal
-- recursively.

begin
  If Ptr /= null then
    InOrder(Ptr.LeftChild);
    WhenTraversingDo(Ptr.Element); -- Perform action on each node.
    InOrder(Ptr.RightChild);
  end If;
end InOrder;

begin
  InOrder(T.Root);
end InOrderTraversal;


Procedure PostOrderTraversal(T : in AVLTree) is
-- pre - T exists.
-- post - Current Location remains unchanged.  Action
--       specified by user supplied WhenTraversingDo
--       procedure is performed on each node in the tree in
--       postorder fashion.

Procedure PostOrder(Ptr : in Node_Pointer) is
-- Hidden procedure that actually performs the traversal
-- recursively.

begin
  If Ptr /= null then
    PostOrder(Ptr.LeftChild);
    PostOrder(Ptr.RightChild);
    WhenTraversingDo(Ptr.Element); -- Perform action on
  end If;                          -- each node.
end PostOrder;

begin
  PostOrder(T.Root);
end PostOrderTraversal;


Procedure LevelByLevelTraversal(T : in AVLTree) is
-- pre - T exists.
-- post - Current Location remains unchanged.  Action
--       specified by user supplied WhenTraversingDo
--       procedure is performed on each node in the tree in
--       a level by level fashion starting at the level
--       containing the root node.

L : List;
Position : Integer;
Ptr, I : Node_Pointer;

begin
  If not Empty(T) then
    Ptr := T.Root;
    Insert(L,1,Ptr);    -- Enqueue root node of tree in L.
    Loop
      If Empty(L) then
        Exit;                -- Traversal finished.
      end If;
```

```
        Delete(L,1,I);        -- Serve first node of list L.

        WhenTraversingDo(I.Element); -- Perform action on
                                     -- each node.
        If I.LeftChild /= null then -- Enqueue next
          Length(L,Position);          level's nodes ...
          Position := Position + 1;
          Insert(L,Position,I.LeftChild);
        end If;

        If I.rightChild /= null then
          Length(L,Position);
          Position := Position + 1;
          Insert(L,Position,I.RightChild);
        end If;                    -- ... left to right.

      end Loop;

      Clear (L);           -- Reclaim memory used by list L.
    end If;
  end LevelByLevelTraversal;
  end Gen_AVLTree;
```

Instantiation of AVL Tree

```
with ada_io;
use ada_io;
with Gen_AVLTree, NodeOutput, IntegerLessThan;
Procedure avltest is
Package Int_AVL is new Gen_AVLTree(Item => Integer,
              WhenTraversingDo => NodeOutput,
              Item1LessThanItem2 => IntegerLessThan);
   -- NodeOutput details can be found in Appendix E.
   -- IntegerLessThan details can be found in Appendix F.

T : Int_AVL.AVLTree;
Int1 : INTEGER;
Int2 : INTEGER;
Bool1 : BOOLEAN;
```

```
ch : Character;

begin
  Loop
    new_line;
    put ("0. Create.");
    new_line;
    put ("1. Full?");
    new_line;
    put ("2. Empty?");
    new_line;
    put ("3. Clear.");
    new_line;
    put ("4. Insert.");
    new_line;
    put ("5. Delete.");
    new_line;
    put ("6. Retrieve.");
    new_line;
    put ("7. Find.");
    new_line;
    put ("8. Traverse.");
    new_line;
    put ("9. Destroy the Tree.");
    new_line;
    new_line;
    put ("Enter number of action you desire.");
    get (Int1);

    Case Int1 is
      when 0 =>
        Int_AVL.Create (T);
        new_line;
        put("Tree created.");

      when 1 =>
        new_line;
        put("Checking if tree is full.");
        If Int_AVL.Full(T) then
          put(" Yes it is full.");
        Else
          put(" No it is not full.");
        end If;
```

```
when 2 =>
  new_line;
  put ("Checking if tree is empty.");
  If Int_AVL.Empty(T) then
    put(" Yes it is empty.");
  Else
    put(" No it is not empty.");
  end If;

when 3 =>
  new_line;
  put("Clearing the tree.");
  Int_AVL.Clear(T);
  put(" Finished clearing the tree.");

when 4 =>
  new_line;
  put("Enter integer you want to put in tree. ");
  get(Int2);
  new_line;
  Int_AVL.Insert(T,Int2,Bool1);
  new_line;
  If Bool1 then
    put("Insertion was successful.");
  Else
    put("Insertion was not successful.");
  end If;

when 5 =>
  new_line;
  put("Enter integer you want deleted from the");
  put(" tree.");
  get(Int2);
  new_line;
  Int_AVL.Delete(T,Int2);
  put(" Finished with deletion.");

when 6 =>
  new_line;
  put("Retrieving content of current node.");
  Int_AVL.Retrieve(T,Int2);
  put(" Content was ");
  put (Int2);
  put (".");

when 7 =>
  new_line;
  put("Enter integer you want to find in the");
  put(" tree. ");
  get(Int2);
  new_line;
  put("Finding ");
  put(Int2);
  put(".");
  Int_AVL.Find(T,Int2);
  put(" Finished with find.");

when 8 =>
  new_line;
  Loop
    put("Enter either 'r', 'n', 'o' or 'e' for ");
    new_line;
    put ("r = pre-order, n = in-order, o =");
    put(" post-order or e =");
    put (" level by level traversal. ");
    get(ch);
    Case ch is
      when 'r'|'R' =>
        Int_AVL.PreOrderTraversal(T);
        Exit;
      when 'n'|'N' =>
        Int_AVL.InOrderTraversal(T);
        Exit;
      when 'o'|'O' =>
        Int_AVL.PostOrderTraversal(T);
        Exit;
      when 'e'|'E' =>
        Int_AVL.LevelByLevelTraversal(T);
        Exit;
      when others =>
        put ("Try again enter either 'r','n'");
        put (",'o', or 'e'.");
        new_line;
    end Case;
  end Loop;
  new_line;
```

```
        put("Traversal finished.");

      when 9 =>
        Int_AVL.Destroy(T);
        new_line;
        put ("Tree destroyed.");

      when others =>
        EXIT;
    end Case;
  end Loop;

  Exception
    when Int_AVL.NOT_FOUND =>
      put ("  What you were looking for in the tree");
      put (" isn't there.");
    when others =>
      Raise;
end avltest;
```

LIST OF REFERENCES

1. MacLennan, Bruce J., Principles of Programming Languages: Design, Evaluation, and Implementation, 2d ed., Holt, Rinehart and Winston, New York, NY, 1987.

2. Wiener, R., Sincovec, R. "Modular Software Construction and Object Oriented Design Using Ada," Journal of Pascal and Ada and Modula-2, Vol. 3, No.2, March/April 1984.

3. Stubbs, Daniel F., Webre, Neil W., Data Structures with Abstract Data Types and Modula-2, Brooks/Cole Publishing Company, Monterey, CA, 1987.

4. Kapur, Deepak, Srivas, Mandayam, "Computability and Implementability Issues in Abstract Data Types," Science of Computer Programming, Vol. v10, Issue n1, February 1988.

5. Buzzard, G.D., Mudge, T.N., "Object Based Computing and the Ada Programming Language," Computer, Vol. 18, No. 3, March 1985.

6. Watt, David A., Wichmann, Brian A., Findlay, William, Ada Language and Methodology, Prentice-Hall International (UK) Ltd, London, England, 1987.

7. Leach, D.M., Satko, J.E., "Implementation Languages for Data Abstractions," Third Annual International Phoenix Conference on Computers and Communications. 1984 Conference Proceedings, IEEE Computer Society Press, Silver Spring, MD, 1984.

8. Department of Defense Military Standard ANSI/MIL-STD-1815A, Reference Manual for the Ada Programming Language, February 17, 1983.

9. Gilpin, Geoff, Ada A Guided Tour and Tutorial, Prentice Hall Press, New York, NY, 1986.

10. Booch, Grady, Software Engineering with Ada, 2d ed., The Benjamin/Cummings Publishing Company Inc., Menlo Park, CA, 1986.

11. Wirth, Niklaus, Algorithms and Data Structures, Prentice Hall Inc., Englewood Cliffs, NJ, 1986.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Professor C. Thomas Wu, Code 52Hq Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	2
4.	Lieutenant Commander Richard N. Britnell 625 Alberta Avenue Cookeville, Tennessee 38501-2667	2